Resilience and sizing in virtualized environments.

Barbe Thystere Mvondo Djob

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêtée ministériel : 25 mai 2016

Présentée par

Barbe Thystere MVONDO DJOB

Thèse dirigée par **Noël DE PALMA**
et codirigéee par **Alain TCHANA**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Résilience et dimensionnement dans des environnements virtualisés.

Thèse soutenue publiquement le **18 Décembre 2020**,
devant le jury composé de :

M. Pascal Felber
Université de Neuchâtel, Rapporteur, Président

M. Willy Zwaenepoel
EPFL & Université de Sydney, Rapporteur

M. Marc Shapiro
Sorbonne Université—LIP6 & Inria, Examinateur

M. Renaud Lachaize
Université de Grenoble Alpes, Examinateur

M. Daniel Hagimont
INPT/ENSEEIHT, Invité

M. Alain TCHANA
ENS Lyon, Co-Directeur de thèse

M. Noël DE PALMA
Université de Grenoble Alpes, Directeur de thèse

"To my grand-fathers and homonym,
Late MVONDO Barthélemy, and
To my grand-mother
NGO NDJOCK ELIZABETH"

Acknowledgments

Fairly speaking, I acknowledge that many people contributed to this dissertation. I want to apologize if I forget to mention the names of the people who helped me throughout my Ph.D. ; it is an honest mistake.

First, I would like to thank my advisors Alain Tchana and Noel De Palma. By their unwavering perfectionism and enthusiasm, they taught me how to do research that matters. They sacrificed a lot of time to correctly follow and support me (and their other students). I hope to become the same source of inspiration for others as you were to me.

I want to thank Daniel Hagimont, Pascal Felber, Marc Shapiro, Renaud Lachaize, and Willy Zwaenepoel for being on my jury. It is an honor to be reviewed by such world-class experts. I'm sure I will learn a lot from your criticism.

I want to thank all the members of the ERODS team in the LIG laboratory, AVALON team in the LIP laboratory, SCALE team in the I3S laboratory, and SEPIA team in the IRIT laboratory. During all various sojourn in these teams, I was amazed by the kindness of everyone. Every encounter was a real step forward towards the realization of this dissertation. I would like to especially thank Muriel Paturel, ERODS's administrative assistant, for withstanding my numerous requests ; you eased my laboratory life.

I want to thank the task force with whom I worked during my Ph.D., Bao Bui, Boris Teabe, Gregoire Todeschi, Kevin Jiokeng, Kevin Nguetchouang, Kouam Josiane, Lavoisier Wapet, Lucien Arnaud, Mathieu Bacou, Mohamed Karaoui, Stella Bitchebe, Tu Dinh Ngoc, and Yuhala Peterson. It was a real pleasure exchanging with each of you. I wish you all a good continuation in your respective works.

I want to thank my family, especially my father Mvondo Mvondo Barthelemy, my mother Ndjock Fleur Nadine, my younger brother Zoua Mvondo, and my younger sister Lingom Mvondo, for all their support during this fantastic journey.

I can't forget to thank my girlfriend, Françoise Carole Ebango Mbesse, for supporting me through this. You were a wonderful support to me.

Lastly, I would like to thank all my friends that supported me throughout this wonderful experience. I am lucky to have such amazing people surrounding and supporting me.

Thank you, everyone.

Résumé

Les systèmes de virtualisations ou hyperviseurs jouent un rôle crucial dans la pile logicielle des plateformes de cloud computing. Leur conception et leur mise en oeuvre ont un impact significatif sur la performance, la sécurité et la robustesse des applications des utilisateurs du cloud. Les hyperviseurs dits de Type-I sont les plus efficaces, car ils offrent une meilleure isolation et de meilleures performances que leur homologue de Type-II. Pour la majorité des hyperviseurs de Type-I actuel (ex., Xen ou Hyper-V), l'hyperviseur s'appuie sur une machine virtuelle privilégiée (pVM). La pVM accomplit des tâches à la fois pour l'hyperviseur (ex., l'administration des VMs) et pour les VMs (gestion des entrées/sorties). Sur les architectures d'accès mémoire uniforme et non uniforme (UMA & NUMA), cette architecture basée sur la pVM pose deux problèmes :

- (1) *le dimensionnement et le placement des ressources de la pVM (CPU + mémoire)* — En effet, un mauvais dimensionnement et placement des ressources de la pVM impacte fortement la performance des applications des VMs. Le problème est complexe, car il existe une forte corrélation entre les besoins de la pVM et les activités des VMs. Les solutions existantes sont soit des approches statiques qui débouchent à un sur/sous dimensionnement ou ne prennent pas en compte le placement des ressources dans une architecture NUMA.
- (2) *la tolérance aux pannes de la pVM* — La pVM étant un composant central, elle représente un élément critique dont la zone de dommage en cas de défaillance est très large. Les approches existantes pour améliorer la tolérance aux pannes de la pVM offrent des faibles garanties de résilience ou génèrent des dégradations importantes.

Cette thèse propose plusieurs modifications à la pVM d'un point de vue architectural et logique afin de traiter les problèmes susmentionnés. Concrètement, cette thèse introduit :

1. Closer, un principe directeur pour la conception d'un OS adapté aux besoins de la pVM. Closer consiste respectivement à ordonnancer et allouer les tâches et la mémoire de la pVM au plus près des VMs cible. Étant une approche dynamique, il masque le besoin de dimensionner la pVM tout en gérant le placement des ressources sur une architecture NUMA avec sa stratégie de localité.
2. Deux nouveaux mécanismes qui réduisent les dégradations du "page flipping" (l'un des protocoles utilisés dans la virtualisation des E/S réseau) lorsqu'elle est utilisée sur une architecture NUMA. En sélectionnant avec soin les pages de la pVM qui seront utilisées lors du "page flipping" en fonction de leur emplacement, ces mécanismes réalisent de meilleures performances que le protocole de virtualisation réseau actuel.

3. Un ensemble de trois principes directeurs (désagrégation, spécialisation et proactivité) et des techniques d'implémentation optimisée pour construire une pVM robuste sans fortement dégrader les performances des applications des VMs.

Nous avons développé des prototypes d'hyperviseurs (en nous appuyant sur l'hyperviseur Xen) qui mettent en oeuvre les principes susmentionnés. Nous validons l'efficacité de nos prototypes en effectuant plusieurs évaluations avec une série d'applications bien choisies. Les résultats obtenus montrent de meilleures performances que les approches de l'état de l'art tout en observant de faibles dégradations de performance.

Cette thèse met en évidence l'importance de la pVM dans un environnement virtualisé et montre qu'elle requiert plus d'attention de la part de la communauté scientifique.

Mots-clés : Virtualisation, NUMA, hyperviseur, pVM, dimensionnement, résilience.

Abstract

Virtual machine monitors (VMMs) or hypervisors play a crucial role in cloud computing platforms' software stack. Their design and implementation significantly impact the performance, security, and robustness of cloud tenants applications. Hypervisors classified as Type-I are the most efficient, since they offer stronger isolation and better performance than Type-II pendant. In most of today's Type-I virtualized systems (e.g., Xen or Hyper-V), the hypervisor relies on a privileged virtual machine (pVM). The pVM accomplishes work both for the hypervisor (e.g., VM life cycle management) and client VMs (I/O management). On uniform and non-uniform memory access (UMA & NUMA) architectures, this pVM-based architecture raises two challenging problems :

- (1) *pVM's resource sizing (CPU + memory) and placement* — Indeed, an inappropriate pVM sizing and resource placement impact guests' application performance. It is a tricky issue since there is a tight correlation between pVM's needs and guest activities. Existing solutions either propose static approaches which lead to over/under-provisioning or do not consider resource placement in NUMA architectures.
- (2) *pVM's fault tolerance* — Being a central component, the pVM represents a critical component with a large blast radius in case of a failure. Existing approaches to improve the pVM's fault tolerance provide limited resilience guarantees or prohibitive overheads.

This dissertation presents several design changes brought to the pVM from architectural and logical perspectives to tackle these problems. Concretely, this thesis introduces :

1. Closer, a principle for designing a suitable OS for the pVM. Closer consists of respectively scheduling and allocating pVM's tasks and memory as close to the target guest as possible. Closer being a dynamic approach, alleviates the need to size the pVM and handles its resource placement in NUMA architectures with its locality strategy.
2. Two new mechanisms that reduce the overhead of page flipping (an efficient scheme used in network I/O virtualization) when used on NUMA architectures. By carefully selecting pVM pages for page flipping depending on their location, the latter mechanisms achieve better performance than the current network virtualization protocol.
3. A set of three design principles (disaggregation, specialization, and pro-activity) and optimized implementation techniques for building a resilient pVM without sacrificing guest application performance.

We build prototypes of pVM-based hypervisors (relying on the Xen hypervisor) that implements all the principles above. We validate the effectiveness of our prototypes by

conducting several evaluations with a series of benchmarks. The results obtained shows better performance than state-of-the-art approaches and low overhead.

This dissertation highlights the critical role of the pVM in a virtualized environment and shows that it requires more attention from the research community.

Keywords : Virtualization, NUMA, hypervisor, pVM, sizing, resilience.

Scientific Publications

Throughout my thesis, I successfully published the following papers. However, this dissertation primarily builds upon the ideas presented in the publications †.

INTERNATIONAL PUBLICATIONS

- **Fine-Grained Fault Tolerance For Resilient Virtual Machine Monitors**†
  Djob Mvondo, Alain Tchana, Renaud Lachaize, Daniel Hagimont, Noel De Palma
  IEEE/IFIP International Conference on Dependable Systems and Networks — *DSN 2020*
- **Closer : A new design principle for the privileged virtual machine OS**†
  Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, Noel De Palma
  IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems — *MASCOTS 2019*
- **Memory flipping : a threat to NUMA virtual machines in the Cloud**†
  Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, Noel De Palma
  IEEE International Conference on Computer Communications — *INFOCOM 2019*
- **When eXtended Para - Virtualization (XPV) Meets NUMA**†
  Vo Quoc Bao Bui, **Djob Mvondo**, Boris Teabe, Kevin Jiokeng, Patrick Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, Noel De Palma
  European Conference on Computer Systems — *Eurosys 2019*

NATIONAL PUBLICATIONS

- **FaaSCache : Système de cache mémoire opportuniste et sans surcoûts pour le FaaS**.
  Kevin Nguetchouang, Lucien Ngale, Stephane Pouget, **Djob Mvondo**, Mathieu Bacou, Renaud Lachaize — *Journées Cloud 2020*
- **Hardware Assisted Virtual Machine Page Tracking**
  Stella Bitchebe, **Djob Mvondo**, Alain Tchana, Laurent Réveillère, Noel De Palma — *COMPAS 2019*
- **SGX Performance Improvement Using A Smart Function Colocation Algorithm**
  Eric Mugnier, **Barbe Thystère Mvondo Djob** and Alain Tchana — *COMPAS 2019*
- **Memory Flipping : How NUMA affects virtual machines ?**†
  Djob Mvondo, Boris Teabe, Daniel Hagimont, Noel De Palma — *COMPAS 2018*

Table of Contents

Acknowledgments i

Résumé iii

Abstract v

Scientific Publications vii

Table of Contents viii

Introduction **1**

1 Background and Motivations **7**

1.1 Key concepts 8

1.1.1 Server virtualization 8

1.1.2 Type-I and Type-II hypervisors 9

1.1.3 NUMA virtualization 12

1.1.4 I/O virtualization 13

1.2 pVM-centric architecture problems 15

1.2.1 Resource sizing & placement of pVM 15

1.2.1.1 Q_1 — pVM sizing 16

1.2.1.2 Q_2 — pVM resource placement 17

1.2.1.3 Q_3 — pVM resource charging 18

1.2.2 pVM resource placement impact on memory flipping 19

1.2.3 pVM fault tolerance 21

2 Addressing pVM resource management on UMA — NUMA architectures **25**

2.1 Closer : Design Principle 27

2.1.1 Resource management 28

2.1.1.1 Resource management for the main container 29

2.1.1.2 Resource management for the secondary container . . . 29

2.1.2 I/O scheduling in the SC 29

2.1.2.1 Packet reception 30

2.1.2.2 Packet emission 32

2.1.3 uVM destruction and migration scheduling in the SC 32
2.1.3.1 uVM destruction 32
2.1.3.2 uVM live migration 33
2.2 Closer : Evaluations 33
2.2.1 Experimental setup and methodology 33
2.2.2 Resource allocation to the MC 34
2.2.3 Locality benefits 35
2.2.3.1 Administrative task improvement 35
2.2.3.2 I/O task improvement 35
2.2.4 Pay-per-use effectiveness 38
2.2.5 All together 39
2.3 Comparison with state-of-the-art techniques 41
2.3.1 pVM management tasks 41
2.3.1.1 Scrubbing 41
2.3.1.2 Live migration 41
2.3.2 I/O virtualization 42
2.3.3 pVM resource billing 42
2.4 Summary 42

3 Mitigating pVM resource placement effects on zero-copy approach 44
3.1 Dedicated page pool for memory flipping 46
3.1.1 Static value for *poolSize* 46
3.1.2 Dynamic value for *poolSize* 47
3.2 Asynchronous memory migration 48
3.2.1 Periodic memory migration 48
3.2.2 Based on the amount of remote memory 48
3.3 Evaluation 49
3.3.1 Experimental setup and methodology 50
3.3.2 Results analysis 51
3.4 Comparison with state-of-the-art approaches 52
3.4.1 I/O virtualization (again) 52
3.4.2 NUMA virtualization 53
3.4.3 Position of our approaches 53
3.5 Summary 53

4 Improving pVM fault tolerance 55
4.1 Xen pVM's — dom0 overview 57
4.2 PpVMM Design 59
4.2.1 Basic idea 59
4.2.2 General fault model 59

4.3 Implementation . . . 60
4.3.1 XenStore_uk FT solution . . . 61
4.3.1.1 Fault model. . . . 61
4.3.1.2 FT solution. . . . 61
4.3.2 net_uk FT solution . . . 64
4.3.2.1 Fault model. . . . 64
4.3.2.2 FT solution. . . . 64
4.3.3 tool_uk FT solution . . . 66
4.3.3.1 Fault model. . . . 66
4.3.3.2 FT solution. . . . 67
4.3.4 Global feedback loop . . . 67
4.3.5 Scheduling optimizations . . . 68
4.4 Evaluation . . . 69
4.4.1 XenStore_uks . . . 70
4.4.1.1 Robustness . . . 70
4.4.1.2 Overhead . . . 71
4.4.2 net_uk . . . 71
4.4.2.1 Robustness . . . 71
4.4.2.2 Overhead . . . 73
4.4.2.3 CPU usage . . . 74
4.4.3 tool_uk . . . 74
4.4.4 Global failure . . . 75
4.4.5 Scheduling optimizations . . . 75
4.5 Related work . . . 76
4.5.1 pVM resilience. . . . 76
4.5.2 Hypervisor resilience. . . . 77
4.6 Summary . . . 77

Conclusion **80**

References **I**

Table of figures **XII**

List of Tables **XV**

Introduction

Context

Cloud computing adoption by companies is continuously growing and represents up to *90%* [1, 35]. Not surprising considering the numerous advantages brought by this service model. Some of these advantages are no property cost, enhanced scalability, and high availability rates [22]. Cloud computing permit companies to focus on the business logic leaving the burden of maintenance to Cloud providers. To ensure their services to their numerous clients while guaranteeing integrity, confidentiality, and performance, Cloud providers rely on **virtualization**. Real cornerstone, virtualization enables the multiplexing of physical server hardware (e.g., memory) between many users while ensuring isolation among them. Cloud providers achieve virtualization via the help of virtualization systems, generally known as **hypervisors**.

A hypervisor is a software that enables the sharing of physical server hardware resources between several entities known as virtual machines (hereafter VM). Hence, it plays a vital role in ensuring the performance and security of applications running inside VMs. Moreover, it must be robust to ensure high availability rates for VMs. Hypervisors classified as Type-I (e.g., Xen [11], Hyper-V [63], etc.) run directly after the hardware layer. Hence, they offer better isolation and performance than those classified as Type-II that relies on the host operating system (e.g., Oracle VM VirtualBox [95], VMWare Fusion [99], etc.). To keep their trusted computing base (TCB) as smaller as possible, Type-I hypervisors commonly rely on a particular VM that has more privileges than others. We denote it the **pVM (privileged VM)**. The pVM is in charge of hosting tools to administrate/supervise VMs and play a role in the multiplexing I/O devices for VMs.

Problematic

Due to its central role in a virtualized environment, managing the pVM is a tricky task. This is due to a strong correlation between its tasks and those of other VMs. Simply put,

the management of the pVM can be divided into two parts. The first is the sizing of pVM in terms of hardware resources (CPU + memory). The second is the resilience of the pVM. Regarding pVM sizing, the main question is **how much resources (CPU + memory) do the pVM needs ?**. A static strategy will result in oversizing (waste of ressources) or undersizing (lack of resources). This is because the pVM needs relentlessly vary with to the load imposed by VMs. Moreover, on non-uniform memory access (NUMA) architectures where the proximity between CPUs and memory impacts latency, a new dimension must be considered : the placement of the pVM resources. This new dimension raises a second question which is : **Should the pVM resources be placed to optimize its and VMs tasks ?**

The placement of pVM resources matters once more when speaking of the **zero-copy** approach, one of the best techniques used in I/O virtualization. Zero-copy consists of exchanging ownership rights on a set of memory pages between the pVM and a given VM, which are later mapped by the hypervisor in their respective address space. This allows a VM to access data aimed at it (e.g., an incoming network packet) without copying from one address space to another. This dissertation reveals a side effect of using the zero-copy approach on a NUMA architecture. Indeed, due to the default pVM resource placement on NUMA architectures, which is to dedicate an entire socket to the pVM, **repeated zero-copy operations incurs transparent and undesired VMs memory pages migrations between NUMA nodes**. This results in transparent remote memory access for VMs' applications, which results in gradual performance degradation.

Regarding resilience, the central role of the pVM makes it a **single point of failure** with a large blast radius (VMs, hypervisor, I/O devices, etc.) in case of a security breach or failure. As a result, efficient policies are needed to react in case of faults of the pVM. Surprisingly, very few research works explore this issue, which is not irrelevant.

This dissertation details three significant contributions that aim at proposing potential solutions to the aforementioned problems while opening some lines of discussion.

Contributions

The first contribution of this dissertation attacks the problem of pVM resource sizing and placement. Existing approaches such as [80] propose static solutions that do not tailor

with the varying load imposed on the pVM. We tackle this problem by proposing **Closer**. Closer is a design principle to implement a suitable OS for the pVM. Closer promotes the proximity with VMs and the utilization of VMs resources. Based on Closer :

- We propose an architecture where the pVM is considered as two logical entities. The first in charge of tasks whose resource needs are static (e.g., VMs monitoring), and the second in charge of tasks dedicated to VMs execution, such as the multiplexing of I/O devices. Each task of the second entity runs as close as possible as the target VMs and uses the latter resources (those of the target VM).
- We revisit Linux to construct an OS for the pVM.
- We prove its efficiency with the Xen virtualization system, a popular open-source hypervisor (used by Amazon AWS).
- We report results obtained with micro- and macro-benchmarks. These results show (1) no resource waste for the pVM resources since we provision pVM resources on-demand, (2) performance improvement for administrative tasks such as (creation, shutdown, migration, etc.), compared to a standard Linux used as the OS of the pVM and Xen as the hypervisor. We improve VMs shutdown and migration times by up to 33%.
- We improve the performance of applications running in VMs, for intensive I/O workloads, up to 36,5% for network packet latency reception, 42,33% for network packet latency emission, and 22% for disks operations.

This dissertation's second contribution attacks the side-effects of the zero-copy approach for I/O in a virtualized system running on NUMA architectures. At the time of writing this dissertation, no work reveals or corrects this issue. We propose two approaches to reduce zero-copy side-effects and offer better performance for VMs simultaneously. Concretely :

- The first approach consists of maintaining a pool of memory pages in VMs, which are strictly reserved for zero-copy operations. Hence, only these pages can become transparently remote, which avoids unwanted remote access for applications in VMs.
- The second approach consists of a task that periodically brings back the VMs memory pages that became remote due to zero-copy to return to the initial memory topology.
- Our evaluations with the Xen virtualization system show that zero-copy side-effects are almost canceled while guaranteeing performance for applications running in VMs.

Finally, the third contribution of this dissertation attacks the resilience issue of the pVM. Despite the pertinence of this issue, it has received very little attention from the community. [41] focused on the reliability of network device drivers hosted in the pVM. However, they do not consider the dependency with other pVM services. [20] propose to

disaggregate the pVM, coupled with a periodic reboot of each disaggregated component (via micro-reboot), to the TCB and prevail against stepping-stone attacks. Unfortunately, periodic reboots incur degradation with up to 1300x order of magnitude, making the solution impraticable in production environments. To tackle this issue, we propose **PpVMM (Phoenix pVM-based VMM)**, an approach that relies on three principles : *disaggregation, pro-activity, and specialization*. The fault model targeted by PpVMM is the failure of the pVM or memory corruption of its stateful components. Concretely :

- PpVMM consists of decomposing the pVM in smaller blocks in charge of a specific task (disaggregation). For each block, dedicated fault detection and correction mechanisms are designed to react to failures (pro-activity + specialization). PpVMM relies on the hypothesis that the hypervisor is fault-free via techniques such as [16, 41, 60].
- We implement a prototype based on PpVMM, with the Xen virtualization system.
- Our prototype's evaluation shows performance degradation of up to 12,7% for the 95 percentiles, with the *TailBench* suite [42]. Nevertheless, it is hugely better compared than existing approaches Xoar [36] that presents performance degradation of up to 12999% for the 95 percentiles with the same benchmark.

The source code of our prototypes is available to the scientific community via this link : https://djobiii2078.github.io/#experience.

Roadmap

The rest of this dissertation is organized as follows :

- Chapter 1 presents the fundamental concepts used in this dissertation. It provides an overview of virtualization and I/O virtualization. It ends presenting of each problem and some insights on why each issue is a significant problem.
- Chapter 2 details our contribution related to the pVM resource sizing and placement. It provides an overview on the related works and reports the evaluation results obtained with our prototypes.
- Chapter 3 details our contribution related to the side-effects of zero-copy on NUMA architectures. It highlights the role of the pVM in this issue and presents the evaluation results obtained.
- Chapter 4 details our contribution related to the pVM resilience. It presents our new architecture with evaluation results where we compare against existing solutions.

Ending this dissertation, a conclusion that resumes our findings, details our current and future works (short- and long- term perspectives) related to the aforementioned problems.

1 Background and Motivations

In this chapter, we present important concepts used in this dissertation (§1.1). Then, we outline challenges encountered with pVM-centric architectures (§1.2). For each challenge, we explain how scathing it is and how prior work tries to address them.

Chapter overview

1.1 Key concepts 8
1.1.1 Server virtualization 8
1.1.2 Type-I and Type-II hypervisors 9
1.1.3 NUMA virtualization 12
1.1.4 I/O virtualization 13
1.2 pVM-centric architecture problems 15
1.2.1 Resource sizing & placement of pVM 15
1.2.2 pVM resource placement impact on memory flipping 19
1.2.3 pVM fault tolerance 21

1.1 Key concepts

In this section, we define the key concepts around server virtualization, NUMA, and I/O virtualization.

1.1.1 Server virtualization

Sharing server physical resources among different entities (applications, operating systems — OSs) while ensuring isolation has always been trendy. Server virtualization comes as a response to the previous need. [12] defines virtualization as *the application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized*. The basic unit of virtualization is a **virtual machine** defined as *a complete compute environment with its own isolated processing capabilities, memory, and communication channels* [12]. Back in the days, in 1964, the term virtual machine was introduced by IBM with the CP/CMS system [21]. As shown in Figure 1.1, the term virtual machine has evolved alongside virtualization techniques, leading to new tools and new virtualization paradigms. Today, we mostly differentiate two types of VMs : *system-level* and *lightweight VMs or containers*. System level VMs allows you the execution of multiple OSs on the hardware of a single physical server. At the same time, containers enable you to deploy multiple applications using the same OS on a single server or system-level VM. Figure 1.2 describes the general architecture for system-level and containers. Each (system-level VMs and containers) comes with their pros and cons. System-level VMs provide stronger isolation than containers while containers, being lightweight (since they rely on a ready kernel) provides better performance due to the absence of an additional software layer between applications and the hardware. Unless specified otherwise, VMs refers to system-level virtual machines.

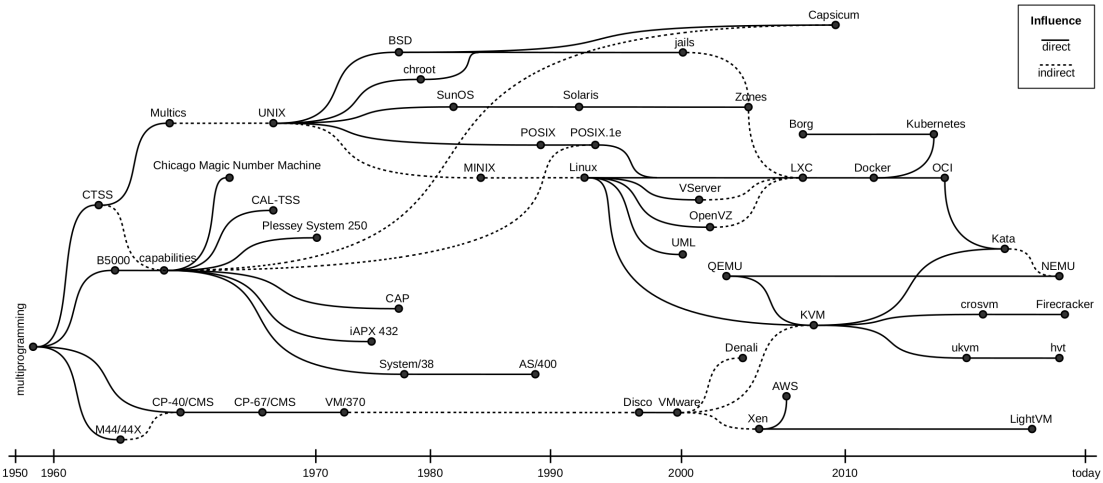

Figure 1.1 – Evolution of virtual machines and containers [82].

In cloud computing, depending on the level of service requested (e.g., Software as a

Service — SaaS) and the cloud provider (e.g., Amazon), services will either be powered by VMs, containers, or both. Much research work tries to find the best of both worlds. [57] leverage unikernels to have VMs boot times faster than containers. However, due to binary compatibility issues with unikernels, the latter scope's work is quite small [1]. On the other hand, works like gVisor [32] and KataContainers [43] propose approaches to strengthen container isolation at the expense of performance [8, 109]. We can see that there's still a lot of interest in VMs.

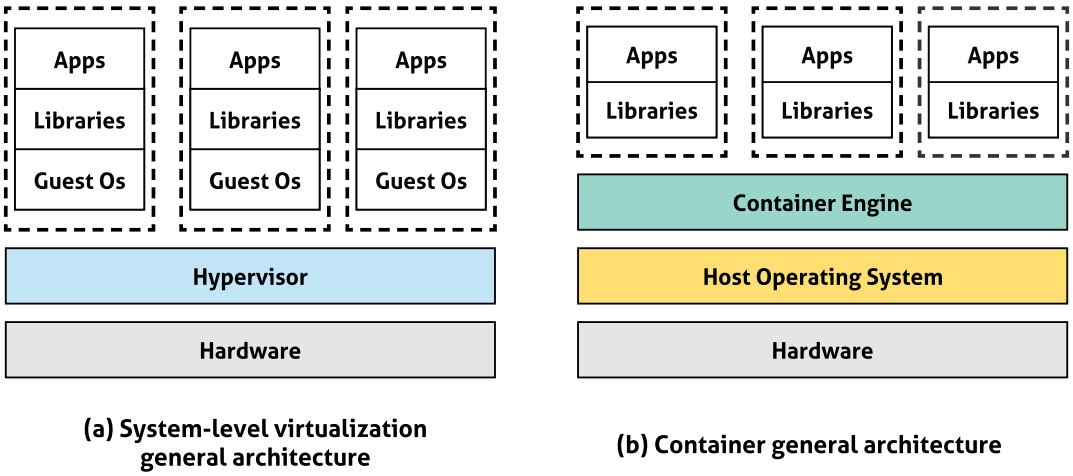

Figure 1.2 – General architecture of system-level virtualization against containers.

To create and manage VMs, we need a **virtual machine monitor (VMM)** or **hypervisor**. A hypervisor is the software layer that will ensure physical resource sharing between VMs that runs guest OSs. Hypervisors are usually classified into two categories : **Type-I and Type-II**.

1.1.2 Type-I and Type-II hypervisors

Figure 1.3 presents the overall architecture of Type-I and II hypervisors. The main difference between Type-I and Type-II hypervisors resides on the privilege level where the hypervisor runs. Type-I or bare-metal hypervisors (e.g., Xen, Hyper-V, VMWare ESX [98], etc.) run directly above the hardware in the ring of highest privilege while Type-II or hosted hypervisors (e.g., VirtualBox, VMWare Fusion, VMWare Workstation [97], etc.) run within an OS along with the Ring of the host OS. The ring privilege level difference between the two categories will affect each type in terms of performance and security, as shown in Table 1.1. Overall, public cloud providers will generally prefer Type-1 hypervisors over Type-II hypervisors due to performance and security issues. Type-II

1. Work such as [74] start providing solutions to unikernel binary compatibiliy

hypervisors will generally be used in environments where performance and security are lesser concerns, such as software testing environments.

Characteristics	Type-I (bare-metal hypervisors)	Type-II (hosted hypervisors)
Performance	Due to a direct access to the hardware, generates lesser overhead.	Due to the presence of an underlying OS, generates greater overhead due to an additonal level of indirection.
Security	Security flaws and vulnerabilities endemic to OSes are absent from bare-metal hypervisors due to no underlying OS. Furthermore, the TCB (Trusted Computing Base) is smaller, thus potentially lesser security issues [14].	Security flaws or vulnerabilities in the host OS can compromise all of the VMs running above it.
Hardware support	Can benefit from hardware virtualization technologies : Intel VT-x [38] or AMD-V [7] to reduce virtualization overhead	

Table 1.1 – Differences between Type-I & Type-II hypervisors based on different parameters : performance, security, and hardware support.

This dissertation focus is on Type-I hypervisors. With most Type-I hypervisors, for simplicity, maintainability, and security purposes, a particular VM is used as an extension of the hypervisor. We refer to this VM as the **privileged VM (pVM)**. The latter VM is critical to the virtualization stack. The pVM embeds unprivileged VMs (uVMs) lifecycle administration tools (e.g., *libxl* in Xen, *parent partition* in Hyper-V) and data center administration applications (e.g *novaCompute* in OpenStack [78]). In addition to these tasks, the pVM is also used in most deployments as a proxy for sharing and accessing I/O devices, see Figure 1.4. In this case, the pVM embeds the driver enabling access to the hardware device and a proxy (called backend), which relays incoming (from the driver to a uVM) or outgoing (from a uVM to the driver) requests. This pVM-based design is popular and used in production-grade, mainstream virtualization platforms(for example, Xen, Microsoft Hyper-V, and some versions of VMware ESX) for several important reasons, including the following ones : (i) it simplifies the development, debugging, and customization of the control plane [11], (ii) it provides isolation boundaries to contain the impact of faults within the control plane or the I/O path (see §1.1.4), (iii) it offers flexibility for the choice of the OS hosting the control plane (which matters for considerations like code

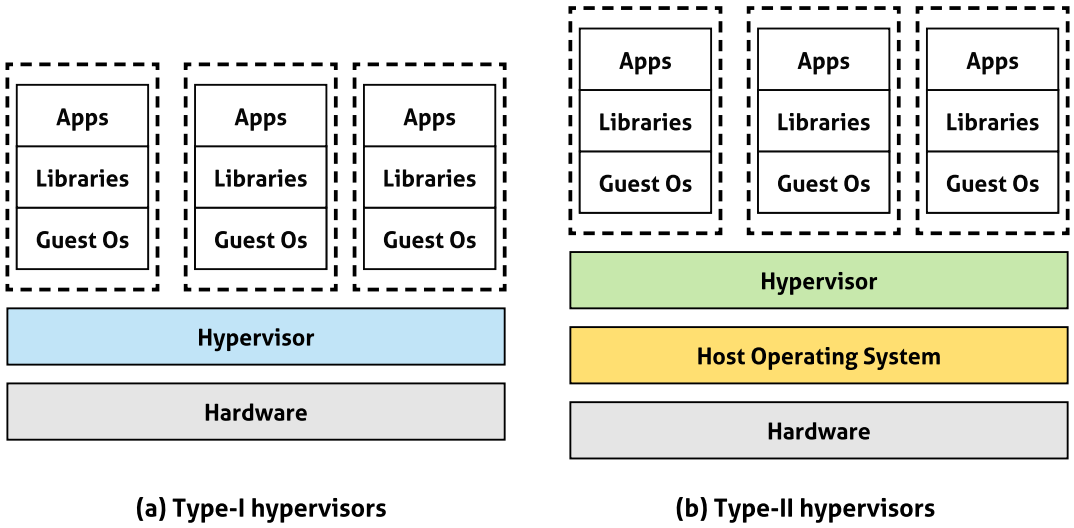

Figure 1.3 – General architecture of Type-I hypervisors (bare-metal) against Type-II hypervisors (hosted).

footprint, security features, and available drivers for physical devices) [87], (iv) it provides a data plane with a smaller attack surface than a full-blown operating system like Linux.

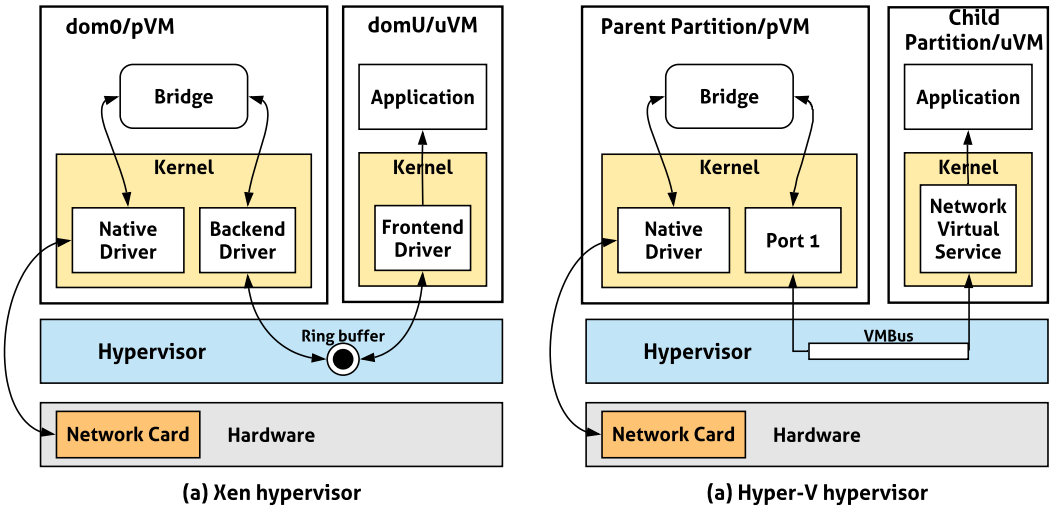

Figure 1.4 – Architecture of two widely used Type-I hypervisors, Xen and Hyper-V. Each has the notion of pVM to support the hypervisor with administrative and uVM I/O management tasks.

1.1.3 NUMA virtualization

In the past, hardware manufacturers designed processors symmetric multiprocessors — SMP or uniform memory access — UMA, where CPU accessed a shared memory over a single bus. However, with the increase of the number of cores (thus CPUs) on commodity servers, the bus rapidly a bottleneck to memory access. To overcome this situation, in modern servers, each CPU is physically connected to its memory module(s), forming a node, and can access remote memory of other nodes in a cache coherent manner via a CPU interconnect [5, 37, 93, 117] as shown on Figure 1.5. CPUs are also equipped with I/O controllers that mediate direct memory access (DMA) by devices to the system's main memory. Remote accesses into a module M are satisfied by the memory controller of M's CPU. Node topologies are such that some nodes might be connected to others indirectly via intermediate nodes, in which case remote accesses traverse through multiple memory controllers.

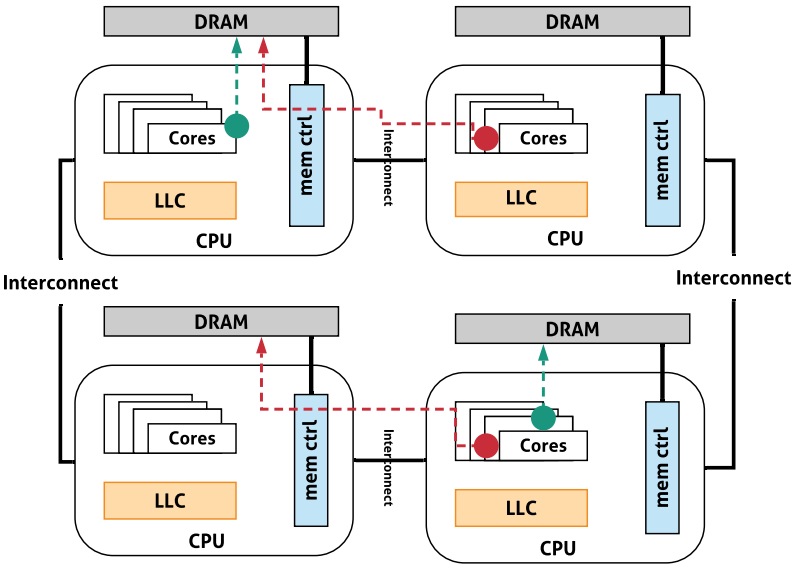

Figure 1.5 – Example of a NUMA architecture. Nodes are linked via Interconnects (such as AMD HyperTransport [7] or Intel QuickPath Interconnect [38]). Local accesses (in green) are faster than remote accesses (in red).

For good performance in NUMA architectures, OSes must ensure that applications always run on the node hosting their data to avoid remote memory accesses. Mainstream OSs embeds NUMA placement and scheduling policies to take advantage of NUMA architectures.

On NUMA architectures, the most common virtualization approach is known as vNUMA [15]. With vNUMA, the hypervisor presents to uVMs a virtual NUMA topology, which corresponds to the mapping of its allocated resources on NUMA nodes at boot time. The OS running inside the uVM can then benefit from its NUMA-aware allocation and sche-

duling policies. To present the virtual topology to uVMs, the hypervisor stores the virtual topology in the uVM's ACPI (*Advanced Configuration and Power Interface*) tables so that the uVM's OS can read them at boot time. However, vNUMA main limitation is that a change in the real NUMA topology (e.g. vCPUs migrations between CPUs) won't be taken into account without rebooting the VM. This is because mainstream OSs do not take into account changes in ACPI tables once booted. Some research work, such as [33, 54, 83, 100], tries to tackle this issue by implementing new policies and context-based heuristics. For now, hypervisors usually disable mechanisms such as memory ballooning or live/cold migration on VMs using vNUMA because they can modify target VMs NUMA topology.

1.1.4 I/O virtualization

A benefit of virtualized systems is the decoupling of a uVM's logical I/O devices from its physical implementation. Decoupling enables time- and space-multiplexing of I/O devices, allowing multiple logical devices to be implemented by a smaller number of physical devices. Applications of virtualization such as server consolidation or running heterogeneous operating system environments on the same machine rely on this feature. Decoupling also enables popular uVM features such as the ability to suspend and resume a virtual machine and the ability to move a running virtual machine between physical machines, known as live migration. In both of these features, active logical devices must be decoupled from physical devices and recoupled when the uVM resumes are saved or moved.

The classic way of implementing I/O virtualization in Type-I hypervisors is to structure the software in two parts : an emulated virtual device that is exported to the uVM — the frontend and a backend implementation that is used by the virtual-device emulation code to provide the semantics of the device (usually hosted in the pVM). This is known as the **split-driver model** [2]. As shown on Figure 1.4 depicts the flow of an I/O request in a virtualized system powered by a typical Type-I hypervisor, Xen. When an application running within a uVM issues an I/O request, typically by making a system call, it is initially processed by the I/O stack in the guest OS within the uVM. A device driver (the backend) in the guest issues the request to a virtual I/O device, which the hypervisor then intercepts. The hypervisor schedules requests from multiple uVMs onto an underlying physical I/O device via the corresponding frontend device driver managed by the pVM with direct access to physical hardware. Upon an I/O request processing completion, the two I/O stacks (pVM and target uVM) are traversed in the reverse order. The actual device posts a physical completion interrupt, which is handled by the hypervisor. The hypervisor

2. The practice of modifying the guest to introduce logic to ease virtualization performance and reduce overhead is known as *para-virtualization*.

then notifies the pVM, which finds the target backend driver [3]. The backend driver via the hypervisor and shared memory initiates data transfer with the target frontend driver in the uVM. Given that the data is stored in memory pages of the pVM, the hypervisor has to give the access grants on these pages to the uVM. To counterbalance the transmission of these pages, the hypervisor gives access grants on some pages from the uVM to the pVM. This mechanism is known as **memory flipping** and allows a **zero-copy memory** communication between the backend and frontend driver. The target guest OS copies the user space's received data and frees the pages containing the data for further usage. Memory flipping provides a significant benefit for I/O applications but as shown n Section §1.2.2, has a major drawback for uVMs on NUMA architectures.

Another technique for I/O virtualization is to use **PCI pass-through mode**. PCI pass-through consists in granting exclusive direct access of an I/O device to a uVM. It is relatively easy to configure the CPU virtualization, so the x86 instructions that talk to the device can be connected directly to the device and incur zero I/O virtualization overheads. This pass-through mode can eliminate both the device emulation and back-end implementation overheads. However, this direct I/O virtualization approach introduces several limitations and implementation challenges. Aside from the obvious limitation that only a single VM can use passthrough device, passthrough forms a coupling between the hardware and the VM. As a result, many of the portability benefits of virtualization are lost, along with key benefits such as live migration and features that depend on the ability to interpose on I/O [85]. This approach also raises strong challenges related to DMA (Direct Memory Access). A guest device driver can program device DMA to read and write memory belonging to the hypervisors or other guests. This can result due to bugs in the device driver or the guest just being malicious. To eliminate both the limitations and passthrough challenges, device builders have modified their hardware to be aware of the virtualization layer. To handle the limitation of exclusive passthrough-only devices, such virtualization-aware hardware exports multiple interfaces, each of which can be attached to a different VM. As a result, each uVM is given its own directly accessible passthrough copy of the device. For example, in Single Root I/O Virtualization — SR-IOV, a NIC can be shared by multiple uVMs while bypassing both the hypervisor and the pVM. However, strong limitations such as no live migration have slowed down its adoption in datacenters [61, 79, 84, 85, 96].

In summary, we can see that bare-metal hypervisors (Type-I) heavily rely on the pVM for an overall good performance. However, this pVM-centric architecture raises some challenges which are tricky. In the section below, we clearly outline each challenge and

3. To reduce overhead, some hypervisors perform virtual interrupt coalescing [4] in software, similar to the hardware batching optimizations found in physical cards, which delay interrupt delivery with the goal of posting only a single interrupt for multiple incoming events.

the related work surrounding the latter.

1.2 pVM-centric architecture problems

Let's recall the role of the pVM in most bare-metal hypervisors. The pVM acts as a control plane for the hypervisor, through a specific interface, and is involved in all uVM management operations (creation, startup, suspension, migration, etc.). It also hosts I/O device drivers that are involved in all I/O operations performed by uVMs on para-virtual devices. The most important characteristic of the pVM is the **correlation between pVM's tasks and uVM's activities**. However, current pVMs rely on standard OSes e.g., Linux. A standard OS (running in a pVM/uVM) manages resources for its applications and itself, but is not aware of resources managed in other VMs running on the same host. Therefore, the previous correlation between pVM and uVMs is not considered in a standard OS, thus leading to resource waste, low performance on NUMA architectures, performance unpredictability, and vulnerability to DoS attacks [36, 71]. This is because the latter correlation makes it difficult to correctly **size** and **place** the resources of the pVM on UMA/NUMA architectures. Lastly, being a central component, the pVM represents a **single point of failure** with a large **blast radius** in case of failure.

Below, we discuss the issues with pVM's resource sizing — placement and fault tolerance. For each issue, we present its impact on the virtualized infrastructure performance.

1.2.1 Resource sizing & placement of pVM

Resource management for the pVM is a tricky task and it can have a significant impact on both administrative tasks and user applications, especially on NUMA architectures which are commonly used in today's datacenters. A pVM resource management strategy must correclty address these questions :

- Q_1 : how much resources (# of CPUs, amount of memory) should be allocated to the pVM ?
- Q_2 : how to organize such an allocation (in terms of location) in a NUMA architecture ?
- Q_3 : in a cloud environment, to who (cloud provider or user) these resources should be charged to ?

Let's discuss each question and present how tricky solving it is.

1.2.1.1 Q_1 — pVM sizing

Our collaboration with some datacenter operators such as Nutanix datacenter [73] revealed that the most common strategy for sizing the pVM is a **static configuration**. This means that the pVM is allocated a fixed amount of resources at startup which do not change throughout its life-cycle. Moreover, virtualization system providers do not provide any recommendation regarding this issue. The only recommendation we found comes from Oracle [80] which proposes only a memory sizing strategy for the pVM based on the following formula : $pVM_{mem} = 502 + int(physical_{mem} \times 0.0205)$. More generally, there isn't any defined method to estimate the amount of resources required by the pVM to perform correctly.

The resources required by the pVM are not constant as they depend on uVM activities, therefore a static allocation cannot be the solution. In fact, the tasks executed by the pVM can be organized in two categories : *tasks related to the management of the datacenter and tasks related to uVM I/O operations*. The amount of resources required by tasks from the second category is mercurial as it depends on uVM activities. A static allocation can lead to two situations, either the pVM's resources are **insufficient** or **overbooked**. These two situations can be harmful for both the cloud provider and user applications. Below, we focus on the consequences of an insufficient ressource provisioning to the pVM because it is obvious that over-provisioning causes resource waste as shown by prior works [92].

A lack of resource in the pVM can make both applications executed in uVMs and administrative services executed in the pVM inefficient and unpredictable. Performance unpredactibility is known as one of the main issues in the cloud [24, 62, 71].

Impact on users' applications

Figure 1.6 top right shows the performance of a uVM which hosts a web application (wordpress) on a physical machine where we vary the number of colocated uVMs which execute I/O intensive workloads. In this experiment, the pVM is allocated two CPUs while each uVM is allocated one CPU. To prevent any contention (e.g. QPI link contention), all CPUs (from the pVM or uVMs) are allocated on the same NUMA socket. Fig. 4 top left presents the pVM CPU load during each experiment. The first observation is that the pVM load varies according to uVMs activities. This is due to the fact that the pVM embeds both the backends and the drivers responsible for accessing I/O devices (see Fig. 3). The second observation is that the web application's performance decreases when the pVM lacks CPU resources. Therefore, performance predictability is compromised.

One may ask if this unpredictability is effectively caused by lack of computation power for the pVM. Since all uVMs execute I/O intensive workloads, the I/O hardware could be the bottleneck. To clarify this point, we ran the same experiment with 12 CPUs for the pVM. The results on Fig. 4 bottom show that with enough resources, the performance of the tested application remains almost constant, which proves that resources allocated to

the pVM are the bottleneck.

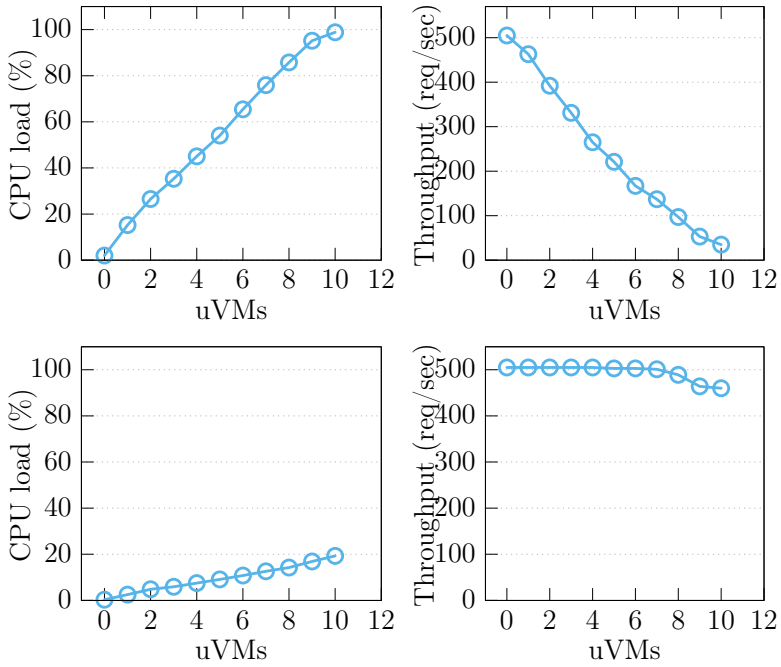

Figure 1.6 – A static resource allocation to the pVM can lead to its saturation, thus to performance unpredictability for user applications : (left) pVM's CPU load, (right) performance of the uVM which executes a (Wordpress) workload. The pVM has 2 CPUs for the top results and 12 CPUs for the bottom.

Impact on management tasks

The pVM hosts VM management operations, the most important ones being : VM creation, destruction and migration. The saturation of the pVM can lead to execution time variation for these operations since they require a significant amount of resources. Figure 1.7 left and right respectively show VM creation and migration times according to the pVM load. We observe that the pVM load has a significant impact on these execution times. This situation may dramatically influence cloud services such as auto-scaling [71].

1.2.1.2 Q_2 — pVM resource placement

On a NUMA machine, the location of the resources allocated to the pVM may significantly influence uVMs performance. The commonly used strategy is to locate all pVM resources on a dedicated NUMA socket, not used by any uVM. This section shows that running the pVM close to uVMs may improve the performance of the latter.

I/O intensive applications' improvement

We executed a web application in a uVM whose entire CPU and memory resources were located on the same NUMA socket as the pVM. Then, we varied the location of the pVM

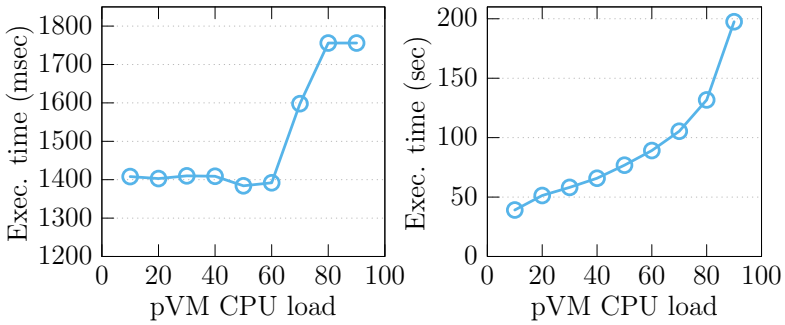

Figure 1.7 – A static resource allocation to the pVM may lead to its saturation, thus to variable uVM (left) creation and (right) migration time.

resources. We observed that the best performance is obtained when the pVM and the uVM share the same NUMA socket (521 req/sec and 8.621 ms latency if colocated vs 491 req/sec and 13.431 ms latency if not). This is because colocation on the same socket prevents remote memory accesses (to fetch I/O packets) from both pVM and uVM tasks.

VM migration and destruction improvement

We also observed that running the pVM close to uVMs may improve some management tasks such as live migration. For instance, we observed in our testbed that the migration of a 2GB RAM uVM can be improved by about 34.15% if the migration process running in the pVM is scheduled on the same NUMA socket which hosts the migrated uVM's memory. We made a similar observation for uVM destruction tasks : the scrubbing step (memory zeroing) is much faster when the migration process runs close to the NUMA socket which hosts the uVM's memory.

1.2.1.3 Q_3 — pVM resource charging

The commonly used strategy is to leave the provider support the entire pVM resources, which includes the resources used on behalf of uVMs for performing I/O operations. [36] showed that this is a vulnerability, which could lead to deny of service attacks.

Overall, we observe that a bad resource management strategy (sizing + placement) for the pVM can severely affect administrative tasks and uVMs applications' performance. There's a need for a strategy that can automatically adapt to the pVM's needs (in terms of size and placement) with a fair charging policy. Furthermore, we discovered a side-effect regarding the memory flipping technique (see §1.1.4) when the pVM resource placement is not taken in account on NUMA architectures. The next section carefully describes the issue.

1.2.2 pVM resource placement impact on memory flipping

As mentioned in § 1.1.4, memory flipping is a mechanism that prevents data copy between backend device driver (in the pVM) and frontend device driver in uVMs during I/O virtualization. Meanwhile, as described in § 1.1.3, when virtualizing a NUMA architecture, the trend (with vNUMA) will be to disable features such as migration on target uVMs since they can disrupt the mapping of target uVMs resources on NUMA nodes.

However, we found that due to the default pVM NUMA resource placement strategy (i.e., dedicating one NUMA node for the pVM — see §1.2.1.2), memory flipping can be the cause of topology modifications for uVMs. Indeed, frequent network communications between the pVM and a uVM (backend $\longleftrightarrow$ frontend) lead to significant exchange of pages, resulting in a displacement of the memory of the uVM from its initial NUMA nodes to the pVM NUMA node and vice versa. Let's note that after frontend drivers release the pages containing the request data, those pages are freed and can be reused by uVMs applications, inducing potential remote accesses for these applications therefore leading to degraded performance. A solution to this issue is to use memory copy the communication between the backend driver and frontend driver. However, numerous memory copies can lead to disastrous performance for I/O applications in uVMs. To assess how harmful, the previous scenarios (memory flipping and memory copy) can be harmful, we conducted the following experiment.

Details about the experimental environment and used benchmarks can be found in Section 3.3.1. The experimental procedure is as follows. Initially, in a *uVM* configured on a single NUMA node, we run the Stream benchmark and measure the throughput (first run). The Stream benchmark measures the memory bandwidth (only) and is very sensitive to remote memory access. This first run corresponds to the reference performance for the benchmark because it does not involve remote memory access and neither memory flipping nor memory copy. Then, we execute Big Bench which is configured to perform I/O operations with another VM on a second server. During the execution of Big Bench, we continuously monitor the memory layout of our *uVM*, and we measure the performance of the benchmark. We run Big Bench because its involve I/O operations, therefore memory flipping or memory copy is used. This creates the pertubation we want to observe in the NUMA topology of the *uVM*. Then, we rerun the Stream benchmark to obtain the new value of the memory throughput (second run). This procedure is executed with vanilla Xen as the guest OS using firstly memory flipping (we call it xen_{flip}), and secondly memory copy (we call it xen_{copy}).

For this experiment, we are interested in four measurements : (1) the memory layout of our *uVM* during the execution of Big Bench, (2) the performance of Big Bench, (3) the throughput of Stream benchmark (first and second run) and finally, (4) the number of remote memory allocation in our *userVM* during the executions of the Stream benchmark.

The obtained results are presented in Fig. 1.8 and Fig. 1.9. From left to right in Fig. 1.8,

we have : the memory layouts of our *userVM*, respectively with memory flipping (a) and memory copy (b), and the performance of Big Bench (c). Fig. 1.9 shows the throughput of the Stream benchmark and the number of remote memory allocations during the executions of the Stream benchmark.

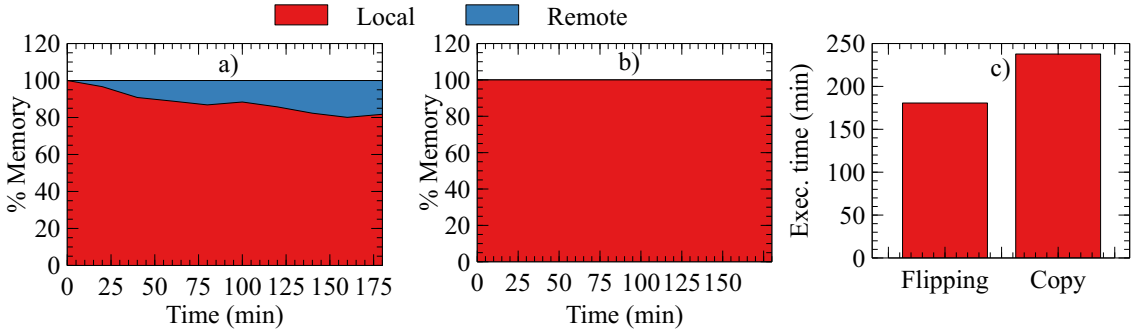

Figure 1.8 – uVM memory layout with memory flipping (a) and memory copy (b) - application performance for Big Bench (c).

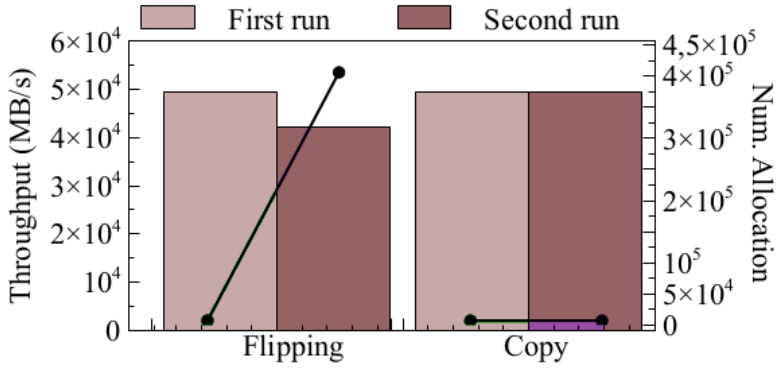

Figure 1.9 – Performance of the Stream benchmark : throughput (left axis / histogram boxes) and number of remote allocations (right axis / solid lines).

The first observation is on Fig 1.8(a) where we simply observe a progressive displacement of our *uVM* memory from its initial node to the *pVM*'s node with memory flipping. After 180 minutes of execution, 25% of the memory of our *uVM* has been relocated to the *pVM*'s node. This displacement is not observed with the memory copy (see Fig 1.8 (b)). However, in Fig 1.8(c), we note that the performance of Big Bench is significantly degraded with memory copy compared to memory flipping. This shows why memory copy cannot be a suitable solution to the problem we address. Fig. 1.9(left) shows the loss in performance for the Stream benchmark with memory flipping, about 13% between the two runs (first and second run). This is explained by the fact that an amount of our *VM* memory which was relocated to the *pVM*'s node (consequently to the execution of Big Bench) is afterward used by the Stream benchmark. The solid lines in the histogram boxes represent the number of remote allocations during the executions of the Stream benchmark (the axis to which it refers to is on the right). There isn't any remote alloca-

tion with memory flipping during the first run, but this is not the case during the second run. This explains the performance degradation for the second run. With memory copy, the number of remote allocation remains void. The main lessons we learn from these experiments are :

- memory flipping can effectively lead to NUMA topology changes for an *uVM* ;
- memory copy is a solution to the problem, but causes significant performance degradation for I/O applications ;
- the displaced memory of a *uVM* can later be used by applications, causing performance degradation.

Finally, after investigating pVM's resource sizing and placement surroundings, we investigate the pVM fault tolerance. Being a single point of failure, it is critical to understand how the pVM can affect the virtualized infrastructure in case of failures and how that can be dealt with. The next section discuss about the pVM fault tolerance.

1.2.3 pVM fault tolerance

pVM-based hypervisors presents two points of failures : the hypervisor and the pVM. While the hypervisor robustness has been hugely investigated by the research community, the pVM has not received the same attention. Surprisingly, its blast radius in case of a failure is ridiculously enormous since it embeds services to manages administrative tasks, multiplex I/O devices (in paravirtualization), and hosts datacenter monitoring components. To assess the problem, we investigate the Xen virtualization system.

Failures within its pVM (dom0 — domain 0) are likely to occur since it is based on Linux, whose code is known to contain bugs due to its monolithic design, large TCB (trusted computing base) and ever-increasing feature set. We analyzed *xen.markmail.org*, a Web site that aggregates messages from fourteen Xen related mailing lists since October 2003. At the time of writing this paper, we found 243 distinct message subjects including the terms *crash, hang, freeze, oops and panic* [4]. After manual inspection of each of the *243* messages, we discarded *82* of them because they were not talking about faults. 57% of the remaining messages were related to failures of the pVM components and 43% to the hypervisor. By zooming on dom0 faults, we observed that 66% were related to device drivers, 26% to the tool stack, and 8% to XenStore (a *procfs* subsystem for storing and sharing uVMs configuration and status informations). From this analysis, two conclusions

4. We used the search string "crash hang freeze oops panic -type=checkins". The option "type=checkins" excludes commit messages

can be drawn : (1) cloud sysadmins report pVM failures ; (2) such failures are linked to all pVM services.

At the time of writing this dissertation, the only existing exhaustive solution against pVM failures (without resorting to physical server replication) is the one proposed in the Xoar project [36]. This approach was initially designed against security attacks, but also provides fault tolerance benefits. It has two main aspects. First, pVM is disaggregated in several unikernels in order to confine each service failure. Second, each service is periodically restarted ("refreshed") using a fresh binary. The critical parameter in such an approach is the refresh frequency. On the one hand, if it is large (tens of seconds), then components that are impacted by a dom0 failure will experience this failure for a long time. On the other hand, if the refresh period is too short (e.g., one second) then failures are handled relatively quickly, but at the expense of significant performance degradation for the user applications. This dilemma has been partially acknowledged by the authors of Xoar in their paper [36] : in the case of a short refresh period (1 second), they measured a 3.5 degradation ratio for the throughput and latency of a Web server benchmark. We also assessed this limitation by running latency-sensitive applications from the TailBench suite [42] in a domU while varying the refresh period of its assigned network backend unikernel (the details of the testbed are provided in §V). Figure 1.10 reports for each benchmark the ratio of the mean and (95th and 99th percentile) tail latencies over the execution of the same benchmark without refresh. We can see that self refresh can incur a 5x-2000x degradation for the mean latency, 5x-1300x for the 95th percentile, and 5x-1200x for the 99th percentile. We also notice that the degradation remains significant even with a large refresh period (60 seconds).

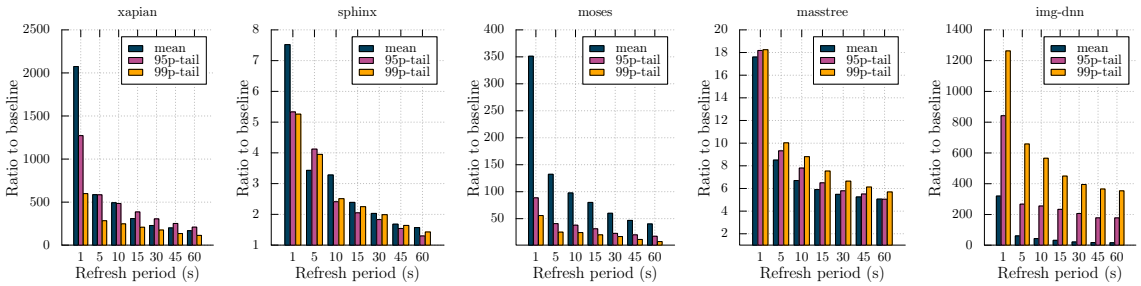

Figure 1.10 – Mean and tail latencies for TailBench applications when self-refresh is enabled for the (disaggregated) pVM components. The results (lower is better) are normalized with respect to a baseline without self-refresh for the same metrics.

As we can see, there are some issues that must be dealt regarding the pVM, regarding resource sizing, placement and fault tolerance. In the next chapters, we will present our contributions to address these issues step by step. For each contribution, we discuss how

it is different from existing approaches and present the different perspectives to improve them.

2

Addressing pVM resource management on UMA — NUMA architectures

In this chapter, we present our contribution that aims at addressing pVM resource management on UMA and NUMA architectures. We detail our design (§2.1) and report evaluation results (§4.4). Lastly, we present a comparative study with state-of-the-art approaches (§2.3).

Chapter overview

2.1 Closer : Design Principle 27
2.1.1 Resource management 28
2.1.2 I/O scheduling in the SC 29
2.1.3 uVM destruction and migration scheduling in the SC 32
2.2 Closer : Evaluations 33
2.2.1 Experimental setup and methodology 33
2.2.2 Resource allocation to the MC 34
2.2.3 Locality benefits 35
2.2.4 Pay-per-use effectiveness 38
2.2.5 All together 39
2.3 Comparison with state-of-the-art techniques 41
2.3.1 pVM management tasks 41

2.3.2 I/O virtualization 42
2.3.3 pVM resource billing 42
2.4 Summary 42

As presented in §1.2.1, the resources of the pVM can hugely impact uVMs' application performance. A good pVM resource management strategy must simultaneously handle *sizing (number of CPUs and amount of memory)* and *placement on NUMA architectures* with respect to uVMs. However, since the pVM relies on a standard OS, its resource management does not consider the correlation between pVM tasks and uVMs activities. A significant part of pVM tasks are correlated with uVMs activities : (1) in terms of quantity, as the amount of resources required by the pVM depends on uVMs activities (especially I/O operations), and (2) in terms of location in a NUMA architecture, as resources allocated for one pVM task (e.g., memory) are likely to be used by correlated uVM activities. In the next section, we introduce **Closer**, a design principle for implementing a pVM which solves the pVM resource management issues reported in §1.2.1.2.

2.1 Closer : Design Principle

To take into account the correlation between the pVM and uVMs, Closer influences the design of the pVM with the three following rules :

- **on-demand**. Resource allocation to the pVM should not be static. Resources should be allocated on demand according to uVMs activities. Without this rule, resources would be wasted if over-provisioned or they would be lacking, leading to performance unpredictability.
- **pay-per-use**. The resources allocated to a pVM's task which is correlated with a uVM activity should be charged to the concerned uVM. Without this rule, the *pVM* would be vulnerable to DoS attacks from a VM executing on the same host, which could use most of the resources from the pVM.
- **locality**. The resources allocated to a pVM's task which is correlated with a uVM activity should be located as close as possible (i.e. on the same NUMA socket) to the uVM activity. This rule allows reducing remote memory accesses in the NUMA architecture.

To enforce the *Closer* principle, we introduce an architecture where the pVM is organized in two containers (see Fig. 2.1) : a *Main Container* (MC) and a *Secondary Container* (SC). Each container is associated with a specific resource management policy, which controls pVM's resource mapping on physical resources. We implemented this architecture by revisiting Linux instead of building a new pVM from scratch. By this way, we propose a ready to use pVM.

The MC is intended to host tasks (i.e. processes) whose resource consumption is constant (i.e. do not depend on uVM activities). Other tasks which depend on the acti-

vity of a uVM are hosted in the SC. More precisely, pVM tasks are organized into four groups : (T_1) OS basic tasks (Linux in our case), (T_2) tasks belonging to the datacenter administration framework (e.g. *novaCompute* in OpenStack), (T_3) VM management tasks (create, destroy, migrate, etc.) and (T_4) I/O management tasks (drivers and backends, see Fig. 1.4). Tasks from T_1 and T_2, and almost all tasks from T_3 (except VM destruction and migration tasks) have a constant resource consumption and are executed in the MC. All other tasks use resources according to uVMs activities and they are executed in the SC.

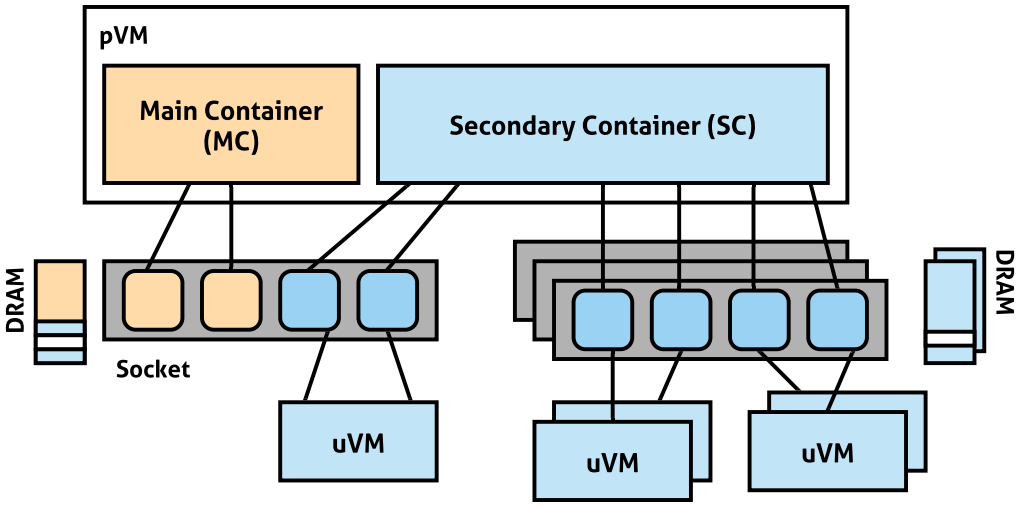

Figure 2.1 – The new pVM architecture.

2.1.1 Resource management

As shown on Figure 2.1, at physical machine startup, the pVM is configured with as many vCPUs as available physical processors (called cores). Each vCPU is pinned on a core (one per core). A subset of vCPUs (therefore of cores) of the pVM is associated with the MC (i.e. used for MC tasks). The rest of vCPUs are linked to the SC and their associated cores are shared with uVMs. Therefore, when a core is reserved for a uVM (the core is the allocation unit), two vCPUs are pinned on that core : the uVM's vCPU and the SC's vCPU associated with that core. This allows the SC to execute the tasks correlated with the uVM on its reserved core (therefore to charge the used resources to the uVM), following the **pay-per-use** and **locality** rules. Regarding the **on-demand** rule, the MC is allocated a fixed amount of resources (vCPU and memory) according to its constant load and the SC is granted a variable amount of resources according to tasks scheduled on its vCPUs.

2.1.1.1 Resource management for the main container

The resources allocated to the MC must be on provider fee, as they are used for executing datacenter administrative tasks (e.g. monitoring). These resources are constant and can be estimated as they only depend on the datacenter administration system (OpenStack, OpenNebula, CloudStack, etc.). Neither the number of VMs nor VMs' activities have an impact on MC resource consumption. Therefore, we use a static allocation for MC resources at physical machine startup and these resources are located on a reduced number of processor sockets. Through calibration, we can estimate the resource to be allocated to the MC. The evaluation section (Section 4.4) provides such estimations for the most popular cloud administration systems.

2.1.1.2 Resource management for the secondary container

The SC includes as many vCPUs as available cores on the physical machine, excluding cores allocated to the MC. At physical machine startup, the SC hosts I/O tasks from the split-driver model (see Fig. 1.4) for uVMs. Even if the I/O tasks are not active at this stage, they require memory for initialization. This initial memory (noted `SCInitialMem` [1]) is very small and assumed by the provider. It will be also used for uVM destruction and migration tasks. Algorithm 1 synthesizes the task scheduling policy in the SC. When a uVM is at the origin of a task (e.g. for an I/O operation), one of its vCPU is scheduled-out and its associated core is allocated to the SC's vCPU mapped to that core, in order to execute this task. The following sections detail the implementation of this algorithm for I/O, destruction and migration tasks.

Algorithm 1 Task scheduling in the SC.

Input : T : an I/O task or VM administration task that should run within pVM
targetuVM=Identification of the target uVM
targetCPU=Pick a CPU which runs targetuVM
Schedule T on the pVM's vCPU which is pinned on targetCPU

2.1.2 I/O scheduling in the SC

pVM's tasks which are executing I/O operations on the account of uVMs are twofold : backends and drivers. The challenge is to identify from the pVM the uVM responsible for each task, in order to use one of its allocated processors for scheduling this task, and to

1. The amount of `SCInitialMem` depends on the number of backend tasks which is bound by the number of vCPU in the SC. We estimated that `SCInitialMem` accounts for about 10MB per backend instance in Xen as an example.

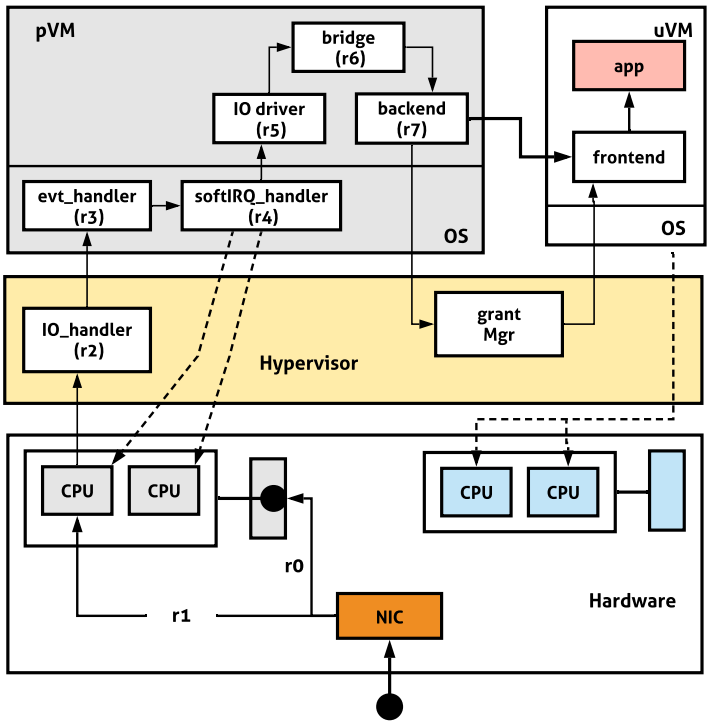

Figure 2.2 – Packet reception workflow.

allocate I/O memory buffers as close to that processor as possible. Given that the split-driver structure is both used for network and disk I/O, we describe in the following our solution for networking tasks which are of two types : packet reception and emission from a uVM.

2.1.2.1 Packet reception

For illustration, let us consider the Xen implementation (see Fig. 2.2), although the description below could be applied to other hypervisors which rely on the split-driver model. When the network adapter receives a packet, (r_0) it places the packet in a queue which was initially allocated in main memory and then triggers an interrupt. (r_1) This interrupt is transmitted to one of the processors of the physical machine by the IOAPIC. (r_2) The interrupt handler which lies in the hypervisor notifies to the pVM the presence of a packet as follows. A vCPU from the pVM (generally vCPU 0) is notified (thanks to the event channel mechanism) and is responsible for reacting to this event. The hypervisor then boosts (prioritizes) this vCPU in its scheduler. When the vCPU is scheduled, it detects the presence of an event and executes the event handler (r_3). This handler generates a softIRQ on one of the vCPU from the pVM. The handler of this softIRQ (r_4) triggers the treatment of the packet which will flow up in the network protocols. There are actually

two ways of treatment : the traditional treatment which works on a per packet basis (we call it OAPI for old API) and a new one (that we call NAPI for new API) which groups message handling in order to reduce the number of interrupts. (r_5) In both cases, the packet has to be copied from the queue to a `skbuff` structure (via the `copybreak` primitive) and the network adapter can then be notified that the packet was well received. The packet is then forwarded to the protocol stack according to the protocol identified in the packet header. In a virtualized system, the destination of the packet is the *bridge*. (r_6) The bridge identifies from the packet header (which includes a target MAC address) the destination backend. The packet then flows down in the protocol stack to the backend. (r_7) On reception, the backend shares with the destination uVM the memory pages which include the packet (this is called *page flipping*) and sends a signal to that uVM. The intervention of the pVM stops here and the packet continues its path in the uVM (starting from the frontend).

In order to implement our *Closer* principle, the general orientation is to force the execution of all r_i steps on one of the processors of the target uVM and to allocate the buffer for the packet on the same socket as that processor.

After step r_0, the incoming packet has been inserted in a queue in main memory and an interrupt triggered on one processor (generally processor 0). The main issue is then to execute the other steps on a processor of the target uVM, while the target uVM is known only at the level of the bridge (step r_6). Regarding step r_1, we rely on *IRQbalance* [53] to balance interrupts between SC's processors (associated with uVMs vCPUs). It does not guarantee that interrupts will be handled by a processor from the target uVM, but it ensures that the MC is not charged for these interrupt handlings, which will be uniformly executed by uVMs processors. This is unfair for uVMs which execute less I/O operations, but the unfairness is limited to step r_1 and mitigated as follows. In the hypervisor, we monitor the I/O activity of each uVM [2] and the notification of a vCPU from the SC (step r_2) is done in proportion to the I/O activity of each uVM (the uVM with the highest I/O activity will receive more notifications on its processors, i.e. SC's vCPUs pinned on these processors will be more solicited). This solution (called hyperLB [3]) is fair, but not precise and inadequate regarding memory management, as the memory hosting the `skbuff` buffer will be allocated on the local socket (of the selected processor from the most I/O loaded uVM) which could be different from the socket of the final target processor. To prevent such a situation, we perform the identification of the target backend (and therefore of the target uVM) in step r_4 (it was previously performed in step r_6 in the bridge) and force the execution of the following steps on one of the processors of that uVM. This solution performs well in the OAPI case where each packet reception generates an interrupt and

2. The monitoring is implemented at the level of the interface between the pVM backend and the uVM frontend.

3. LB stands for load balancing.

the execution of all the steps for each packet. In the NAPI case, a single interrupt can be generated for the reception of a group of packets (with different target uVMs), whose treatment relies on a poll mechanism. We modified the poll function within the driver (we did it for *e1000*, this is the only non generic code) in order to identify the target uVM for each polled packet. In the rest of the document, we call NAPI-1 the per group implementation while NAPI-n refers to the per packet implementation.

2.1.2.2 Packet emission

The out-going path is the reverse of the receive path presented in Fig. 2.2. Applying our *Closer* principle is in this case straightforward since the uVM responsible for the I/O activity is known from the beginning step. We simply enforce the hypervisor to notify the vCPU of the SC associated with a processor of the uVM. Then, we modified the pVM's scheduler for the following steps to be executed on the same processor. The same implementation is used for disk operations.

2.1.3 uVM destruction and migration scheduling in the SC

uVM destruction and migration are administration tasks which are also executed in the SC. Logically, resources consumed by these tasks should be charged to the provider. In our solution, this is effectively the case for memory (with the allocation of `SCInitialMem` described above), but not for CPU which is charged to the concerned uVM. This is not a problem as the goal is here to remove the uVM from the host, and it has a main advantage : the proximity with the memory of the uVM.

2.1.3.1 uVM destruction

The most expensive step in the destruction process of a uVM is memory scrubbing [71]. This step can be accelerated if the scrubbing process executes close to the uVM memory (following the **locality** rule). However, the current implementation consists of a unique process which performs that task from any free vCPU from the pVM. Our solution is as follows. Let S_i, $i \leq n$ be the sockets where the uVM has at least one vCPU. Let S'_j, $j \leq m$ be the sockets which host memory from the uVM. For each S_i, a scrubbing task is started on a vCPU from the SC, the processor associated with this vCPU being local to S_i and shared with a vCPU of the uVM. This task scrubs memory locally. The uVM memory hosted in a S'_j which is not in S_i is scrubbed by tasks executing on other sockets (this remote scrubbing is balanced between these tasks).

2.1.3.2 uVM live migration

uVM live migration mainly consists in transferring memory to the destination machine. The current implementation consists of a unique process which performs the task from any free vCPU from the pVM. The transfer begins with cold pages. For hot pages, the process progressively transfers pages which were not modified between the previous transfer and the current one. An already transferred page which was modified is transferred again. When the number of modified pages becomes low or the number of iterations equals five, the uVM is stopped and the remaining pages are transferred. Then, the uVM can resume its execution on the distant machine. We implemented a distributed version of this algorithm which runs one transfer process per socket hosting the uVM's memory (similarly to the destruction solution), thus enhancing **locality**. These processes are scheduled on SC's vCPUs associated with free processors if available. Otherwise, the processors allocated to the uVM are used.

2.2 Closer : Evaluations

This section presents the evaluation results of our revisited Linux and Xen prototype.

2.2.1 Experimental setup and methodology

Servers. We used two Dell servers having the following characteristics : two sockets, each linked to a 65GB memory node ; each socket includes 26CPUs (1.70GHz) ; the network card is Broadcom Corporation NetXtreme BCM5720, equidistant to the sockets ; the SATA disk is also equidistant to the sockets. We used Xen 4.7 and both the pVM and uVMs run Ubuntu Server 16.04 with Linux kernel 4.10.3. Unless otherwise specified, each uVM is configured with four vCPUs, 4GB memory and 20GB disk.
Benchmarks. We used both micro- and macro-benchmarks, respectively for analyzing the internal functioning of our solutions and for evaluating how real-life applications are impacted. Table 2.1 presents these benchmarks.
Methodology. Recall that the ultimate goal of our proposed pVM architecture is to respect the three rules of *Closer*, presented in Section 2.1 : on-demand resource allocation (to avoid resource waste and enforce predictability), locality (to improve performance), and pay-per-use (to prevent DoS attacks). As a first step, we separately demonstrate the effectiveness of our architecture for each principle. Notice that the evaluation of a given

Table 2.1 – The benchmarks we used for evaluations.

Type	Name	Description
Micro	*Memory*	A memory intensive application - live migration evaluation. It builds a linked list and performs random memory access as in [94]. It has been written for the purpose of this article.
	Netperf [34]	Sends UDP messages - network evaluation. The performance metric is the average request latency.
	dd	Formatting a disk file - disk evaluation. It is configured in write-through mode. The performance metric is the execution time.
Macro	*Wordpress* [103]	Website builder - network evaluation. We used the 4.8.2 version. The performance metric is the average request latency.
	Kernbench [44]	Runs a kernel compilation process - disk evaluation. We compiled Linux kernel 4.13.3. Data caching is disabled. The performance metric is the execution time.
	Magento [56]	ECommerce platform builder - both network and disk evaluation. We used the 2.2.0 version. The performance metric is the average request latency.

principle may not include all contributions. Therefore, a final experiment is realized with all the contributions enabled. For each experiment, we compare our solution with the common pVM's resource allocation strategy (referred to as Vanilla pVM, *Vanilla* for short). In the latter, a set of resources, which are located at the same place, are dedicated to the pVM. Otherwise specified, we dedicate a NUMA socket and its entire memory node to the pVM, corresponding to an oversized allocation.

2.2.2 Resource allocation to the MC

In our solution, MC's resources are statically estimated by the provider. We estimated MC's resources for the majority of cloud management systems namely : OpenStack Ocata [78], OpenNebula 5.2.1 [77] and Eucalyptus 4.4.1 [72]. For each system, we relied on the default configuration and measured its resource consumption. The results, based on our testbed, are presented in Table 2.2. We can see that very few resources are needed for performing all MC's tasks. Our experiments also confirmed that MC's needs do not depend on the IaaS size.

Table 2.2 – MC's needs for three cloud management systems.

	OpenStack	OpenNebula	Eucalyptus
# vCPUs	2	2	1
RAM (GB)	2	1.85	1.5

2.2.3 Locality benefits

We evaluated the benefits of locality on both administrative (uVM destroy and uVM migration) and I/O tasks. Recall that we provide locality by enforcing the execution of SC's vCPUs on the same socket as the concerned uVM. In order to only evaluate locality benefits, SC's vCPUs do not use uVM's CPUs (unless otherwise specified).

2.2.3.1 Administrative task improvement

The effectiveness of our uVM destruction solution is obvious and has been well demonstrated in [71] (We do not claim this innovation and we recommend to refer to [71]). We focus on the novel multi-threaded migration solution we propose, whose effectiveness is not obvious due to the management of dirty pages. To this end, we evaluated our solution when the migrated uVM runs an intensive *Memory* benchmark. We experimented with different uVM memory sizes (4GB-20GB). For every experiment, the uVM's memory is equally spread over the two NUMA nodes of the server. We considered two situations (noted C_1 and C_2) : in C_1, migration threads do not share the uVM's CPUs (assuming there are free CPUs available on the socket for migration) while they do in C_2 (assuming there aren't any available free CPU, the socket which runs the uVM being fully occupied). Fig. 2.3 presents the evaluation results (lower is better). We can see that our solution outperforms *Vanilla* in all experimented situations. The improvement is most important with large uVMs (up to 33% in C_1, see Fig. 2.3). Intuitively, one can imagine that the improvement will also increase when the number of sockets hosting the uVM's memory increases. C_1 slightly outperforms C_2 (up to 4%), justifying our strategy to only use the uVM's CPUs when there is no free CPU on the involved sockets.

2.2.3.2 I/O task improvement

We evaluated the benefits of running pVM's tasks that are involved in a uVM's I/O activity on the same socket as that uVM. We evaluated both network and disk activities. Concerning the former, we evaluated each version presented in Section 2.1.2, namely : *hyperLB* at hypervisor level ; OAPI, NAPI-1 or NAPI-n at the pVM level. For these evaluations, we used a constant low load (100 req/sec).

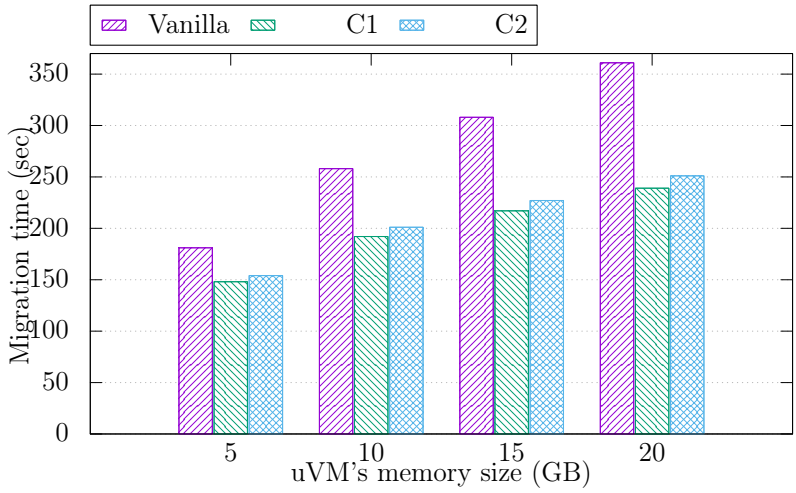

Figure 2.3 – Multi-threaded migration process (lower is better). The uVM runs a memory intensive application.

Packet reception with a single uVM

The uVM is configured with 4vCPUs (all located on the second socket) and 4GB memory. Another server runs *Netperf*. We are interested in the following metrics : (t_1) the treatment duration before the backend invocation (this includes r_{3-6} steps, see Fig. 2.2) ; (t_2) the time taken by the backend to inform the frontend (step r_7) ; and (t_3) the time taken by the frontend to transfer the packet to the application's buffer. Fig. 2.4 left presents the evaluation results of each t_i while varying packet size. Here, all versions provide almost the same values, thus they are shown under the same label (*Our-sol.*) in Fig. 2.4 left. We can see that our solution minimizes all t_i in comparison with *Vanilla*, leading to a low $t_1 + t_2 + t_3$ (up to 36.50% improvement).

Packet reception with multiple uVMs

The previous experiments do not show any difference between our implementation versions. The differences appear only when the server runs several uVMs. To this end, we performed the following experiment. The testbed is the same as above in which a second uVM (noted d_2, the first one is noted d_1) runs on the first socket (the same which runs the MC). The two uVMs receive the same workload. We are interested in d_1's performance and we compare it with its performance when it runs alone (as in the previous experiments). Fig. 2.5 left presents $t_1 + t_2 + t_3$ normalized over *Vanilla*. We can see that OAPI and NAPI-n outperform NAPI-1 by about 21%. This is because NAPI-1 (which does a batch treatment) fails in choosing the correct target uVM for many request in r_4. This is not the case neither in OAPI nor in NAPI-n which are able to compute the correct destination for each packet. Fig. 2.5 right presents the amount of errors (wrong destination) observed for each implementation version. Thus, in further experiments, "our solution"

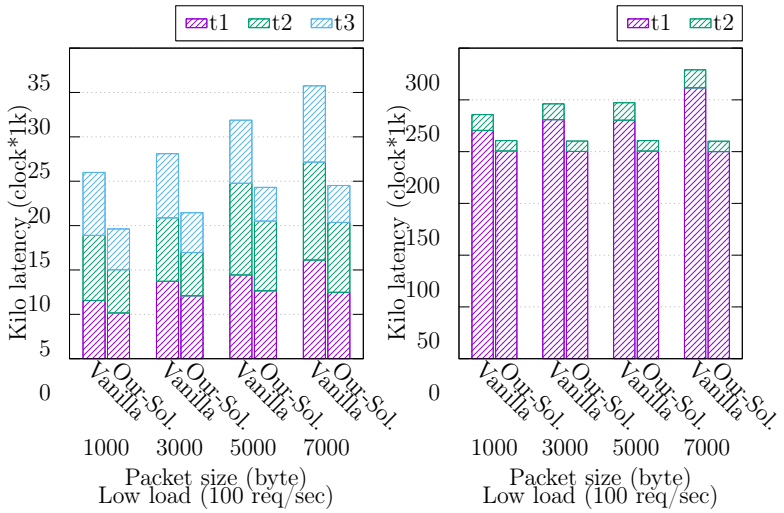

Figure 2.4 – Packet reception (left) and packet emission (right) improved by locality. (lower is better)

refers to OAPI or NAPI-n.

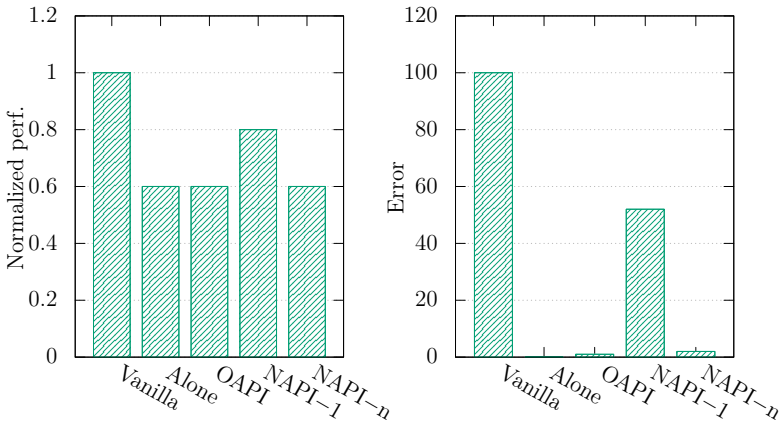

Figure 2.5 – Packet reception improved by locality : multiple uVMs evaluation. Results are normalized over Vanilla (lower is better).

Packet emission

The testbed here is the same as in the experiment with packet reception and a single uVM, except the fact that *Netperf* runs inside the uVM. Remember that packet emission is agnostic about the different implementation versions of our solution (see Section 2.1.2). We are only interested in t_1 and t_2 (t_3 is constant). Fig. 2.4 right presents the results. We can see that thanks to locality, our solution improves (minimizes) t_1 by up to 42.33% for large packets in comparison with *Vanilla*. Compared with packet reception results, the improvement is more significant here because in the case of packet emission with *Vanilla*, memory is allocated (on emission) on the uVM socket and pVM's I/O tasks access the

packet remotely. This is not the case with packet reception since the packet enters the machine via the pVM's socket (therefore, packet handling is more important in the pVM than in the uVM).

Disk operations

The testbed here is the same as above except the fact that the benchmark is *dd*. The uVM's hard disk is configured in write-through mode in order to avoid caching. The collected metric is the execution time. We evaluated different write block sizes. Fig. 2.6 presents the results. We can see that our solution outperforms *Vanilla*, especially with large blocks (up to 22% improvement).

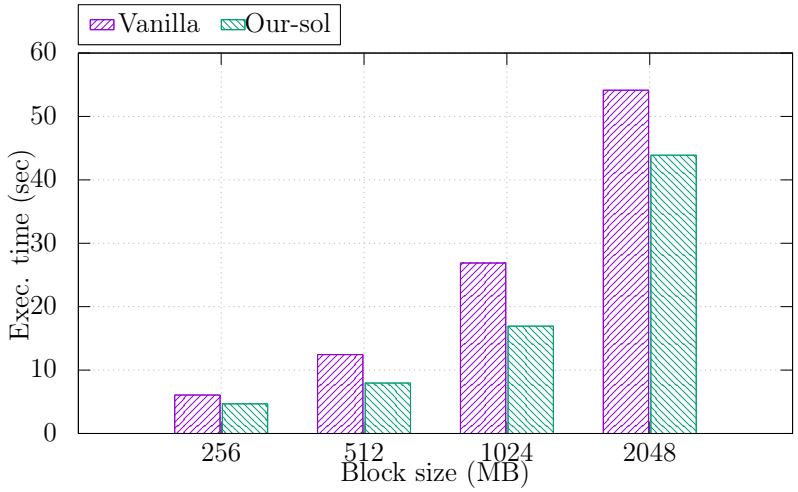

Figure 2.6 – Disk operations (from dd) improved by locality. (lower is better).

Macro-benchmark improvement

We also evaluated the benefits of locality on macro-benchmarks. The testbed is the same as above in which the benchmark is replaced by a macro-benchmark. Fig. 2.7 left presents the results normalized over *Vanilla*. We can see that all the benchmarks are improved with our solution : about 25% for wordpress, 21% for Kernbench, and 30% for Magento.

2.2.4 Pay-per-use effectiveness

We validated the effectiveness of our architecture in charging to uVMs resources consumed by the pVM on their behalf. This includes demonstrating that MC's resource consumption remains constant regardless uVMs activities. We also evaluated the fairness of our solution, meaning that each uVM is charged proportionally to its activity. We only present the evaluation results for the packet reception experiment (which is the most sensitive one, packet emission and disk operations being less tricky regarding pay-per-use). To this end, we used the same testbed as previously. The server under test runs three uVMs (noted vm_1, vm_2, and vm_3), each configured with two vCPUs. They share the same socket

and each vCPU is pinned to a dedicated CPU. vm_1, vm_2, and vm_3 respectively receives 1000req/sec (which accounts for 10% of the total load), 3000req/sec (30% of the total load) and 6000req/sec (60% of the total load). In this experiment, we do not care about memory locality. We collected the following metrics : MC's CPU load, and the proportion of CPU time stolen by SC's vCPUs from each uVM (this represents the contribution of the uVM, used for fairness evaluation).

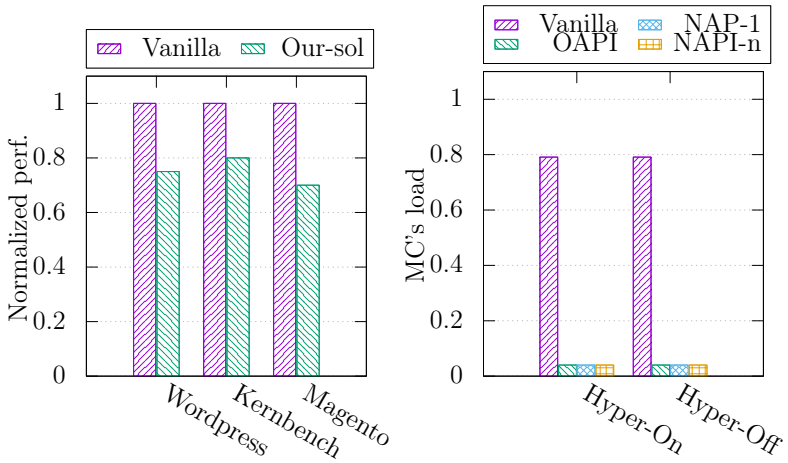

Figure 2.7 – (left) Macro-benchmarks improved by locality. The results are normalized over Vanilla (the first bar). (right) MC's CPU load.

Fig. 2.7 right shows MC's CPU load consumption while Fig. 2.8 presents each uVM contribution. In Fig. 2.7, the ideal solution is the one which leads to no CPU consumption in the MC, meaning that the MC is not impacted by uVMs activities. The load reported for *Vanilla* is relative on the MC reserved capacity (e.g. 100% for *Vanilla* means that its CPU consumption is equivalent to the entire MC capacity). We can see that all our implementation versions are close to the ideal value, the best version being OAPI/NAPI-1 combined with *hyperLB-On* while the worst one is NAPI-n with *hyperLB-Off*. But the difference between these versions is very low, meaning that the activation of *hyperLB* is not necessary. In other words, r_3's CPU consumption is negligible. Therefore, in Fig. 2.8, we only present results with *hyperLB-Off* (*hyperLB-On* provides the same results). The ideal solution is the one which provides the expected result (the first bar). The latter corresponds to the proportion of the uVM's I/O traffic in the total traffic generated by all uVMs. We can see that except NAPI-1, both OAPI and NAPI-n ensure fairness.

2.2.5 All together

We evaluated our solution when locality and pay-per-use mechanisms are enabled at the same time. The testbed is the same as above where macro-benchmarks replace micro-benchmarks. We tested all possible colocation scenarios. Our solution is compared with

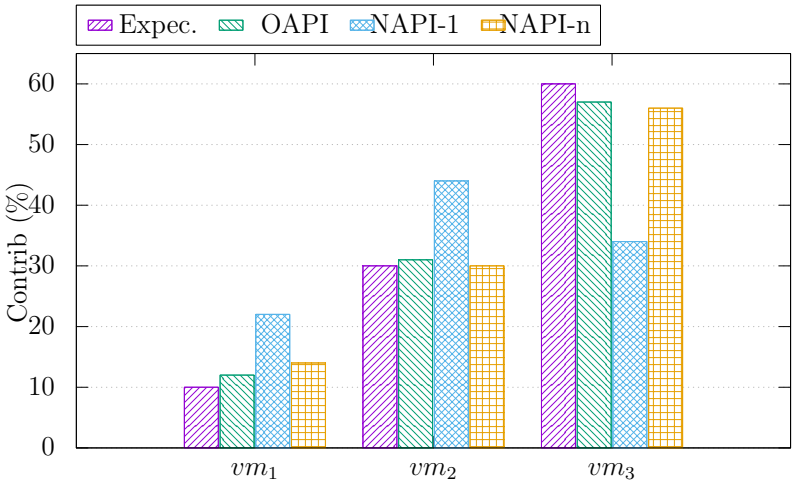

Figure 2.8 – Fairness of our solutions. (close to the first bar is better).

Vanilla when the pVM is allocated the same amount of resources as the MC. Given a benchmark, the baseline value is the one obtained when it runs alone. Having already discussed the impact on performance, the interesting subject here is *performance predictability*. Several studies [31, 59, 92] showed that this issue may occur when the pVM is saturated due to uVMs activities. We also compared our solution with [92] (called *Teabe* here). The latter presented a solution which ensures that the aggregated CPU time (including the one generated inside the pVM) used by a uVM cannot exceed the capacity booked by its owner. Fig. 2.9 presents the results for Kernbench using the best implementation version (NAPI-n with *HyperLB-Off*), all benchmarks running at the same time. In Fig. 2.9, each evaluation <situation>-<solution> is done in a given *situation* and with a given *solution*. Situation can be *alone* or *col* (for colocation of the 3 benchmarks) and Solution identifies the used solutions (*all* is our full solution). We can see that *Teabe* enforces predictability as our solution : in Fig. 2.9 *alone-all* is almost equal to *col.-all*; and *alone-Teabe* is almost equal to *col.-Teabe*. However, our solution improves application performance thanks to our locality enforcement contribution : *alone-all* (respectively *col.-all*) performs better than *alone-Teabe* (respectively *col.-Teabe*). As mentioned above, the improvement will increase when the number of sockets hosting the uVMs memory increases. Fig. 2.9 also shows the results reported in the previous sections so that the we can easily appreciate on the same curve the impact of each contribution.

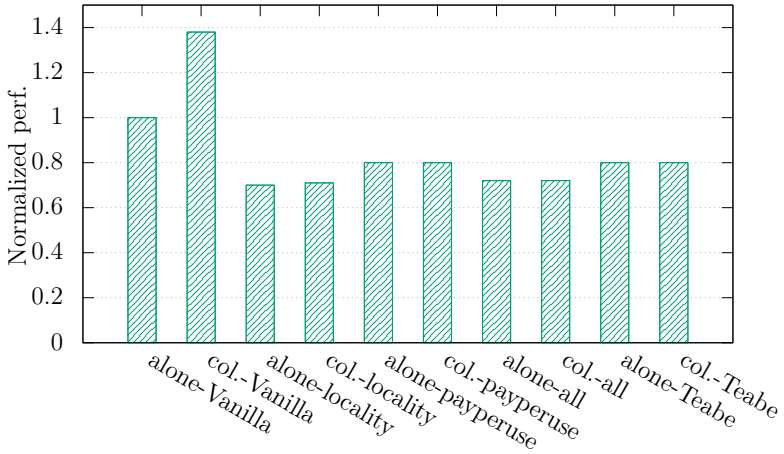

Figure 2.9 – Our solution with all mechanisms, Kernbench results. The latter are normalized over Vanilla when the benchmark runs alone. For predictability, compare <alone>-<solution> and <col.>-<solution>. For performance improvement, lower is better.

2.3 Comparison with state-of-the-art techniques

2.3.1 pVM management tasks

Several research studies have investigated pVM management task improvements. Most of them have focused on uVM creation [45, 46, 48, 55, 57, 71, 81, 112, 113, 116] and migration [2, 39, 47, 70, 88, 102] in order to provide reactive applications (quick elasticity) and consolidation systems. Very few of them exploit locality to improve these management tasks as we do.

2.3.1.1 Scrubbing

Related to our scrubbing optimization, a Xen patch [105] was proposed to delay the scrubbing process and perform it during idle CPU cycles. Contrarily to our solution in which scrubbing is done synchronously with uVM destruction, [105] has the drawback of letting a completely arbitrary amount of time be spent before the memory is available again, introducing a lot of non-determinism. [71] presents a solution which is quite similar to the one we proposed.

2.3.1.2 Live migration

Concerning live migration improvement, we didn't find any solution using a multi-threaded algorithm in order to execute in parallel and close to the migrated uVM's memory as we do.

2.3.2 I/O virtualization

Regarding I/O virtualization improvement, existing solutions either act at the hypervisor scheduler level [3, 75, 91, 107, 108, 110] in order to minimize the scheduling quantum length thus minimizing I/O interrupt handling latency.

2.3.3 pVM resource billing

Very few researchers studied pVM resource utilization (on behalf of uVMs) from the pay-per-use perspective. To the best of our knowledge, [31] and [92] are the only studies which investigated pVM's resource partitioning as we do. [31] is limited to mono-processor machines while [92] leads to resource waste.

2.4 Summary

We identified several issues related to the design of the pVM in today's virtualization systems. These issues arise from the fact that current pVMs rely on a standard OS (e.g., Linux) whose resource manager does not consider **the correlation between pVM's tasks and uVM's activities**. These issues lead to resource waste, low performance, performance unpredictability, and vulnerability to DoS attacks.
To take into account this correlation between virtual machines, we introduce *Closer*, a principle for implementing a suitable OS for the pVM. *Closer* promotes the proximity and the utilization of uVMs resources. It influences the design of the pVM with three main rules : **on-demand** resource allocation, **pay-per-use** resource charging and **locality** of allocated resources in a NUMA architecture. We designed a pVM architecture which follows *Closer* and demonstrated its effectiveness by revisiting Linux and Xen. An evaluation of our implementation using both micro- and macro-benchmarks shows that this architecture improves both management tasks (destruction and migration) and application performance (I/O intensive ones), and enforces predictability. However, it will be interesting to perform more evaluations under more stressful network conditions.

3

Mitigating pVM resource placement effects on zero-copy approach

In this chapter, we present our contribution that aims at addressing pVM resource placement effects on zero-copy (memory flipping) on NUMA architectures. We propose two approaches (§3.1 and §3.2) and report evaluation results (§3.3). Lastly, we present a comparative study with state-of-the-art approaches (§3.4).

Chapter overview

3.1 Dedicated page pool for memory flipping 46
3.1.1 Static value for *poolSize* 46
3.1.2 Dynamic value for *poolSize* 47
3.2 Asynchronous memory migration 48
3.2.1 Periodic memory migration 48
3.2.2 Based on the amount of remote memory 48
3.3 Evaluation 49
3.3.1 Experimental setup and methodology 50
3.3.2 Results analysis 51
3.4 Comparison with state-of-the-art approaches 52
3.4.1 I/O virtualization (again) 52

3.4.2 NUMA virtualization 53
3.4.3 Position of our approaches 53
3.5 Summary 53

As stated in §1.2.1.2, in a NUMA architecture, the trend is to grant an entire NUMA node to the pVM to isolate its resources from those of uVMs. However, in this configuration, memory flipping tends to disrupt the NUMA topology of uVMs and result in performance degradation. To cope with this situation, we propose two solutions to reduce the impact of memory flipping on a VM running on a NUMA architecture. The solutions we propose have been implemented within Xen hypervisor but can be easily integrated into other hypervisors implementing the vNUMA option.

3.1 Dedicated page pool for memory flipping

With the current implementation within Xen, after a flipping operation, *uVM* memory pages that are relocated on the *pVM* node can be subsequently used by applications after the uVM OS frees them. Our first solution aims to limit the number of memory pages that can be remote for an uVM and prevent these pages from being used by applications. At the startup of an *uVM*, we create a pool of pages dedicated to memory flipping. Let's call *poolSize* the size of the pool. Other applications in the *uVM* cannot use these pages. After a memory flip, pages of *uVM* relocated on the *pVM* node are not freed anymore but are instead added to the pool to replace the pages that have been yielded to the *pVM* to will be used for subsequent flips and will be relocated on the *pVM* node. Thus, our solution limits the number of pages that can become remote for the *uVM* and also ensures that any other application cannot use these remote pages.

Fig. 3.1 illustrates our solution's functioning in a simple scenario with the *pVM* running on node 1 and an *uVM* running on node 2. At step (1), all the pages of our *uVM* are on their initial node, and some pages are placed in the page pool for memory flipping. After a memory flip on a single page in step (2), a page of *uVM* is now on the *pVM* node. But this page cannot be used by any application in the *uVM* because it is part of the page pool dedicated to memory flipping. In step (4), after two successive memory flips, we can observe that only pages in the pool are used for flipping, limiting the number of pages from the *uVM* that can be remote. A key point of this solution is the *poolSize* value. We can use two approaches to define this value for a VM : the first is static while the second is dynamic.

3.1.1 Static value for *poolSize*

This approach consists of assigning a static value to *poolSize* for a VM. This value is obtained by calibration. The latter is done by running several I/O applications in a VM and monitoring for each application the number of pages needed for the memory flipping. Then, we can use the highest number of pages required as the value of *poolSize*. This

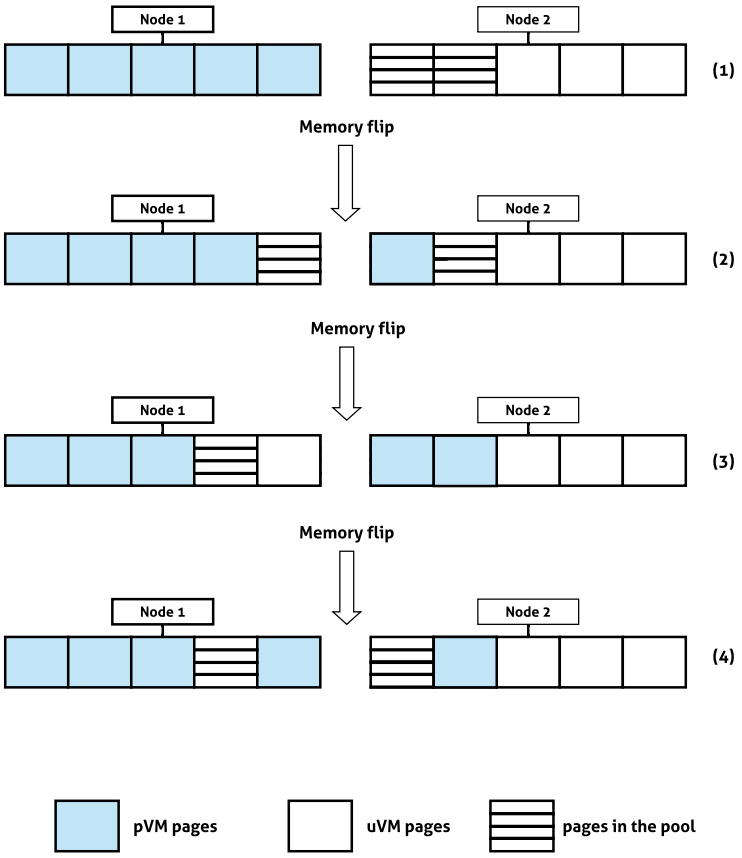

Figure 3.1 – Illustration of the dedicated page pool for memory flipping.

approach has two big disadvantages : (1) calibration is a fastidious task which requires a lot of time and (2) with new application types appearing every day in the cloud, we cannot ensure that a single value of *pooSize* will always be suitable, hence the need for a dynamic solution.

3.1.2 Dynamic value for *poolSize*

The dynamic approach, which we advocate, consists of setting initially an arbitrary value (we call it $poolSize_{init}$) for the *poolSize* of a VM. The value of $poolSize_{init}$ must be lower than the VM's total. Then, during the execution of an I/O application in the VM, *poolSize* is adjusted according to the activity. Algorithm 2 presents the algorithm used to adjust *poolSize*. At each memory flip, we compute the pool percentage, which is used (line 1). If the percentage is greater than 90%, we increase the pool size by 10% (line 3). And similarly, if the percentage is less than 20%, we decrease the pool size by 10%. 90% and 20% are thresholds we obtained after conducting experiments to decide which are suitable to determine when to decrease and increase the *poolSize*.

Algorithm 2 Dynamic poolSize estimation algorithm

```
1: compute used_poolSize
2: if used_poolSize > 90% then
3:     add 10% of poolSize
4: else if used_poolSize < 20% then
5:     remove 10% of poolSize
6: end if
7: set new poolSize value
```

3.2 Asynchronous memory migration

This solution repatriates the remote pages of an *uVM*, by migrating them asynchronously to their initial node. Initially, we let memory flipping behave normally. But at a given time, this solution starts a process that migrates the pages that have been relocated on the *pVM* node back to the *uVM* nodes. This allows the *uVM* to recover its initial topology and avoid remote memory access for running applications. Fig. 3.2 presents a simple example of how this solution works on two nodes. In step (1), all the memory of the *uVM* is on the initial node. In step (2) and (3) we have two consecutive flips on memory pages. In step (4), our solution migrates the pages of the *uVM* to their initial node, which allows recovering the initial topology. Defining the frequency and the condition in which the memory migration process has to be started is the main problem with this solution. Two approaches can be used : the first is periodic, and the second based on the amount of *uVM* memory turned remote.

3.2.1 Periodic memory migration

In this approach, we set a period (we call it $period_{flip}$) which defines when remote pages of an *uVM* will be relocated back to their initial nodes. The administrator can set $period_{flip}$ when starting the VMs. A big value is not suitable because the amount of remote memory can become huge before the migration process gets activated. In contrast, a small value can lead to significant overhead. In the evaluation section, we give more details about the order of magnitude for $period_{flip}$.

3.2.2 Based on the amount of remote memory

In this approach, we launch the migration process when an amount of remote memory (we call it $remote\_memSize$) is reached for an *uVM*. The administrator sets $remote\_memSize$ at the startup of a VM. In the evaluation section, we give more details about the order of magnitude of $remote\_mem$.

Independently of the design used, the migration process is handled by the *pVM* resources.

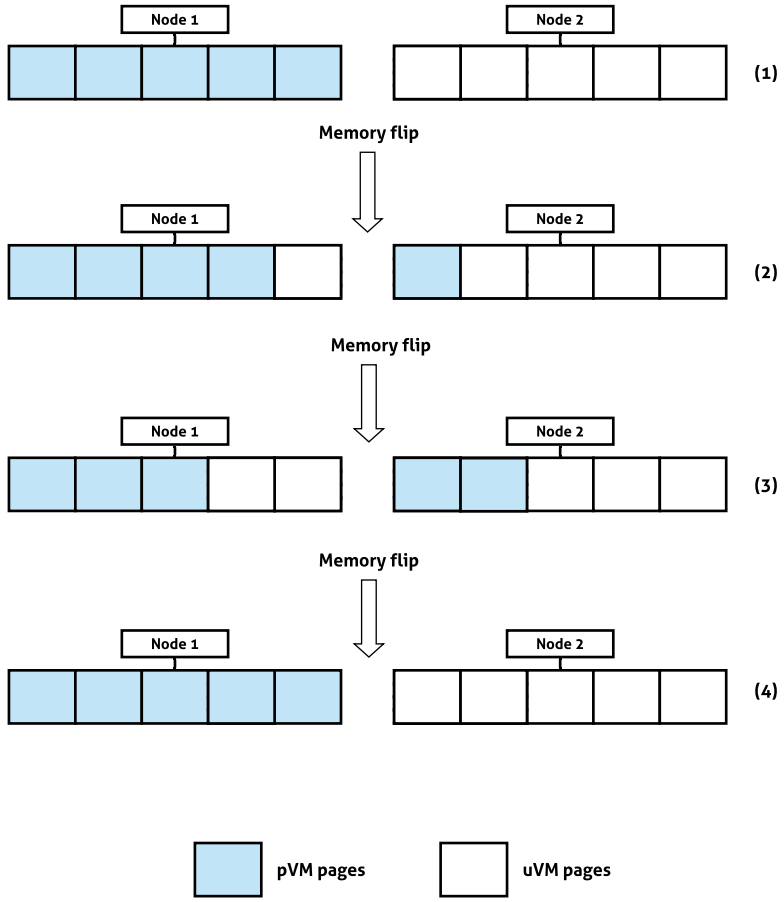

Figure 3.2 – Illustration of the asynchronous memory migration.

3.3 Evaluation

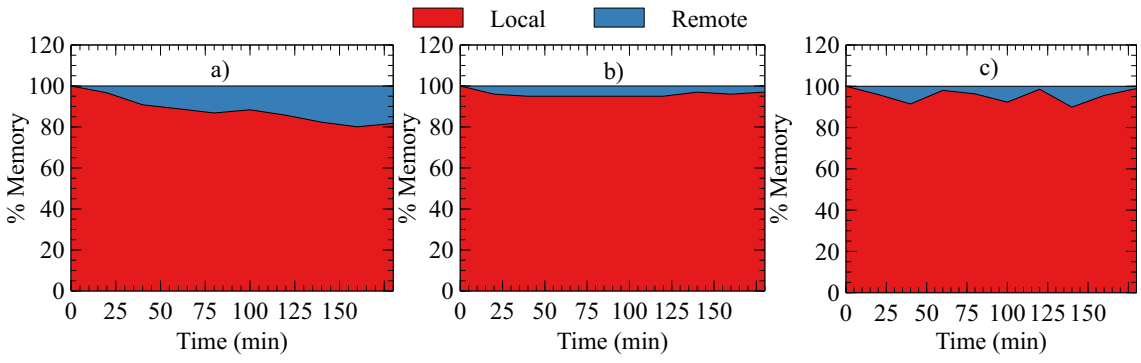

Figure 3.3 – uVM memory layout after running Big Bench : (a) for vanilla Xen, (b) for the page pool solution and (c) for the asynchronuous memory migration

This section presents the evaluation results of our solutions. We implemented prototypes of our solutions in Xen 4.7 [11].

3.3.1 Experimental setup and methodology

Servers. We used two Dell servers with the following characteristics : two sockets, each linked to a 65GB memory node ; each socket includes 26CPUs (1.70GHz) ; the network card is Broadcom Corporation NetXtreme BCM5720, equidistant to the sockets ; we used Xen 4.7 and both *pVM* and *uVM* run Ubuntu Server 14.04 with Linux kernel 4.10.3. Otherwise specified, we configure each *uVM* to use vNUMA option with 16 vCPUs and 16GB of memory on a single NUMA node and 20GB of disk ; the *pVM* has a dedicated NUMA node for its execution.

Benchmarks. We used well known macro-benchmarks for analyzing the impact of memory flipping on the NUMA topology of an *uVM* and evaluate its impact on application performance.

Table 3.1 – Benchmark descriptions

Name	Description
Big Bench. [30]	BigBench is an open-source big data benchmark suite. Big Bench proposes several benchmark specifications to model five important application domains, including search engine, social networks, ecommerce, multimedia data analytics and bioinformatics. The metric used for this benchmark is the execution time.
LinkBench. [10]	LinkBench is a database benchmark developed to evaluate database performance for workloads similar to those of Facebook's production MySQL deployment. The metric used for this benchmark is the execution time.
Stream benchmark. [58]	Stream is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernels. The metric used for this benchmark is the memory bandwith.

Configurations of our solutions. We start with the page pool solution. In our evaluation, knowing all the obvious disadvantages of the static approach (see Secion 3.1), we decided to use only the dynamic approach to set the value of *poolSize* (see Secion 3.1). The value of $poolSize_{init}$ is set to 500MB during the evaluation.

For asynchronous memory migration, the results we present are those obtained with the second approach (the migration process is started when a given amount of remote pages is reached, see Secion 3.2). We observed that with the best value for $period_{flip}$

(900sec) and *remote_memSize* (500MB), the benchmarks have the same performance. Therefore, we decided to present only the results of the second approach with *remote_memSize* set to 500MB.

Methodology. Our evaluation aims to show that :

- our solutions can limit the amount of memory that becomes remote ;
- application performance is not impacted when our solutions are used.

The experimental procedure we used is similar to the one in Section 1.2.2. The procedure was repeated with vanilla Xen (with memory flipping), and with both of our solutions. Initially, we run the Stream or LinkBench benchmark and measure the performance (first run). Then, we execute Big Bench which is configured to perform I/O operations with another VM on a second server. During the execution of Big Bench, we continuously monitor the memory layout of our *uVM*. Then, we rerun the Stream or LinkBench benchmark to obtain the new performance level (second run). This procedure is executed with vanilla Xen and both of our solutions.

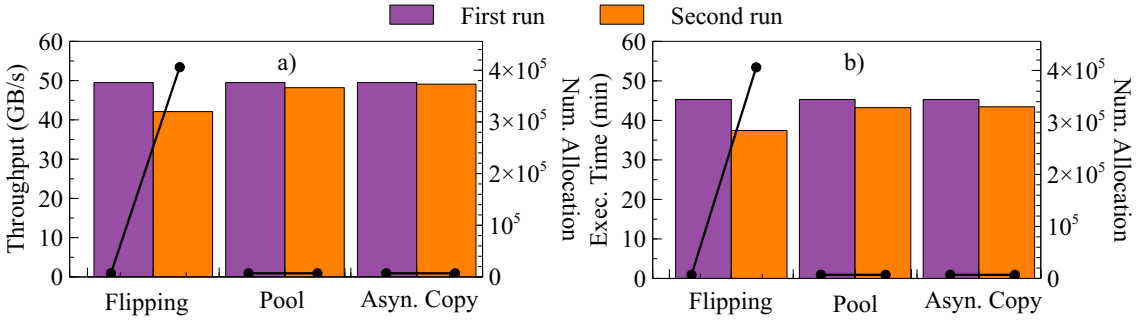

Figure 3.4 – (a) Stream and (b) LinkBench benchmark results.

3.3.2 Results analysis

Fig. 3.3 and Fig. 3.4 show the results of these experiments. Fig. 3.3 presents the memory layout of the *uVM* during Big Bench execution. We observe that with vanilla Xen, after 180 minutes of execution, about 25% of the memory of the *uVM* was moved to another node (Fig. 3.3(a)), thus modifying its NUMA topology. With both of our solutions, we observe that the amount of displaced memory is kept very small. This can be observed in Fig. 3.3(b) (page pool) and (c) (memory migration). Only a very little percentage (3%) of the memory is remote. Fig. 3.4 (a) and (b) present respectively the results for the Stream and LinkBench benchmarks, and the number of remote allocations. The solid lines in the histogram boxes of Fig. 3.4 represent the number of remote allocations during the executions (the axis to which it refers to is on the right). We can observe that the throughputs obtained with our solutions for the Stream benchmark (Fig. 3.4 (a)) are very close for the first and second run (the higher the better). This is not the case with vanilla Xen which has

a performance degradation of about 13%. There isn't any remote allocation with memory flipping during the first run of the Stream benchmark, but this is not the case during the second run. This explains the performance degradation of the Stream benchmark. With our solutions, the number of remote allocation remains constant and nul. We observe a similar behavior with LinkBench in Fig. 3.4 (b).

3.4 Comparison with state-of-the-art approaches

Several research studies have investigated improvements of I/O operations and NUMA handling in virtualized environements. This state of the art is organized according to these two topics.

3.4.1 I/O virtualization (again)

The integration of I/O devices in virtualized environments has always been a subject of primary importance. Many research works investigated the improvement of I/O operations' performance. We can distinguish three main categories of work according to the implementation level : (1) hardware-level works which focus on hardware modification to improve the I/O performance [61], (2) hypervisor level works which propose new scheduling approaches to reduce the latency of I/O requests [108, 111] and finally, (3) guest OS-level works which focus on the optimization the software layer for the I/O processing in the *uVM* and the *pVM* [107]. Works on memory flipping and memory copy techniques are in this third category. The introduction of para-virtualization with Xen (which is the flagship) pushed the hypervisor designers to propose more efficient I/O architecture, leading to the split driver model. The latter allows isolation for fault tolerance while offering interesting performance compared to full virtualisation. The first implementation of the split driver model within Xen was based on the memory copy. But the performance of applications with this approach was very low and had to be improved. Therefore, Xen introduced memory flipping, which ensures zero memory copy during the processing of I/O operations. Several works have studied the split driver model and the frontend/backend communication mechanism [17, 107]. The most interesting of these works proposes establishing a shared memory region between all the VMs for implementing front-end/back-end communication [17]. This approach seems attractive because it provides much better performance for I/O applications than memory flipping and avoids the problem we solve. However, a critical limitation of this approach, which is significant considering that we are in the context of the Cloud, is security. Indeed, establishing a shared memory region between all the VMs breaks VM isolation and opens a gateway to many exploits from malicious hackers.

3.4.2 NUMA virtualization

NUMA management is ubiquitous in both virtualized and non-virtualized environments. Since NUMA architecture's appearance, many scientific works have focused on handling NUMA in non virtualized environments [23, 52]. Generally, studies on NUMA in virtualized environments are simple translations of solutions in non-virtualized environments [100]. The main question that researchers had to answer is where NUMA specificities should be handled : in the hypervisor or in the guest OS. Most hypervisors (Xen, VMWare, and Hyper-V) have adopted a simple solution called vNUMA. The latter consists of presenting the VM's NUMA topology to the guest OS and preserving this topology (by disabling operations that would modify it). This solution is straightforward because it allows the guest OS to benefit from all the already implemented kernel optimizations for NUMA. However, some studies demonstrated that this approach is not suitable for some applications requiring a dynamic NUMA approach [100].

3.4.3 Position of our approaches

At the time of writing this dissertation, we are the first work which addresses the problem of the compliance between vNUMA and memory flipping, which are techniques used by almost all hypervisors. The only existing solution is that of Xen which relies on memory copy, thus degrading I/O performance in VMs. Our solutions keep the locality benefits in NUMA architectures while also maintaining good performance for I/O applications.

3.5 Summary

We observed that after a significant number of memory flips, a substantial part of the memory of *uVMs* is moved to the node of the *pVM*, thus modifying their NUMA topology and forcing these VMs to make remote memory access. This situation hurts application performance in the VMs. We proposed two solutions that attempt to enforce a static NUMA topology for VMs despite memory flipping. We have implemented and evaluated our solutions in the Xen hypervisor. The evaluation shows that our solutions allow us to get close to a static topology for VMs, limiting remote memory access and providing better performance than vanilla Xen for I/O applications.

4

Improving pVM fault tolerance

In this chapter, we present our contributions regarding the pVM fault tolerance. We present PpVMM, which relies on 3 major principles (§4.2). Next, we present some implementation details (§4.3) and report evaluation results obtained (§4.4)

Chapter overview

4.1 Xen pVM's — dom0 overview 57
4.2 PpVMM Design 59
4.2.1 Basic idea 59
4.2.2 General fault model 59
4.3 Implementation 60
4.3.1 XenStore_uk FT solution 61
4.3.2 net_uk FT solution 64
4.3.3 tool_uk FT solution 66
4.3.4 Global feedback loop 67
4.3.5 Scheduling optimizations 68
4.4 Evaluation 69
4.4.1 XenStore_uks 70
4.4.2 net_uk 71
4.4.3 tool_uk 74
4.4.4 Global failure 75
4.4.5 Scheduling optimizations 75

4.5 Related work 76
4.5.1 pVM resilience. 76
4.5.2 Hypervisor resilience. 77
4.6 Summary 77

At this level, we perceive the vital role of the pVM in most Type-I hypervisors. The pVM hosts critical services to administrate and manage uVMs. For this reason, the pVM represents a single point of failure and is an excellent target for security attacks. Moreover, the pVM commonly relies on a standard mainstream OS (e.g., Linux), which has a larger code base than hypervisors (which a higher growth rate — see Table 4.1). For this reason, the pVM is more bug-prone than hypervisors. pVM failures can result in the inability to connect to the physical server, manage user VMs, and interrupt networked applications running in user VMs. Surprisingly as mentioned in §1.2.3, pVM fault tolerance has received little attention from the research community, and existing approaches [20] result in unacceptable overhead (up to 1300×). In the next sections, we present **PpVMM (Phoenix pVM-based VMM)**, a holistic fault tolerance solution for the pVM. We use the Xen virtualization as a case study. PpVMM assumes the hypervisor layer is reliable (via solutions such as [16, 60, 115]). But before the main dish, let's give a rapid overview of Xen's pVM (the dom0) services.

	Xen Hypervisor	**Linux-based pVM**	$\frac{Linux}{Xen}$
#LOC in 2003	187,823	3.72 Million	19.81
#LOC in 2019	583,237	18.5 Million	31.76
$\frac{\#LOC in 2019}{\#LOC in 2003}$	3.11	4.98	1.6

Table 4.1 – Evolution of source code size for the Xen hypervisor and a Linux-based pVM. LOC stands for "Lines of Code". For Linux, we only consider `x86` and `arm` in the `arch` folder. Also, only `net` and `block` are considered in `drivers` folder (other folders are not relevant for server machines). We used Cloc [6] to compute the LOC.

4.1 Xen pVM's — dom0 overview

The dom0 is a Linux system that hosts an important portion of the local virtualization system services, namely (i) the domU life-cycle administration tool (`xl`), (ii) XenStore, and (iii) I/O device drivers. The `xl` tool stack [106] provides domU startup, shutdown, migration, checkpointing, and dynamic resource adjustment (e.g., CPU hotplug). XenStore is a daemon implementing a metadata storage service shared between VMs, device drivers, and Xen. It is meant for configuration and status information rather than for large data transfers. Each domain gets its path in the store, which is somewhat similar in spirit to the Linux `procfs` subsystem. When values are changed in the store, the appropriate

components are notified. Concerning I/0 devices, dom0 hosts their drivers and implements their multiplexing, as follows. Along with I/O drivers, dom0 embeds proxies (called *backend* drivers) that relay incoming events from the physical driver to a domU and outgoing requests from a domU to the physical driver. Each domU runs a pseudo-driver (called *frontend*), allowing to send/receive requests to/from the domU-assigned backend (see Figure 1.4).

Table 4.2 summarizes the negative impact of the failure of dom0 with respect to each service that it provides. We can see that both cloud management operations (VM start, stop, migrate, update) and end user applications can be impacted by a dom0 failure. Concerning the former, they can no longer be invoked in case of dom0 failure. Regarding user applications, those which involve I/O devices become unreachable in case of dom0 failure. Table 4.2 also shows that XenStore (XS) is the most critical dom0 service because its failure impacts all other services and user applications.

	VM manangement operations (impact the cloud provider)				Application operations (impact cloud users)		
	Start	Stop	Migrate	Update	Net I/O	Disk I/O	CPU/ Mem
Tools	A	A	A	A			
Net					A	S	
Disk						A	S
XS	A	A	A	A	S	S	S

Table 4.2 – Impact of the failure of the different dom0 services (`xl` tools, network/disk drivers, XenStore) on the VM management operations and on the applications (in user VMs). An “A” *mark indicates that the failure* always *leads to unavailability while a* “S” *mark denotes correlated failures that occur* only in specific situations.

4.2 PpVMM Design

This section presents the basic idea behind PpVMM and the general fault model that we target.

4.2.1 Basic idea

Our solution, named PpVMM (Phoenix pVM-based VMM), is based on three main principles. The first principle is *disaggregation* (borrowed from Xoar [20]), meaning that each dom0 service is launched in an isolated unikernel, thus avoiding the single point of failure nature of the vanilla centralized dom0 design. The second principle is *specialization*, meaning that each unikernel embeds a FT solution chosen explicitly for the dom0 service that it hosts. The third principle is *pro-activity*, meaning that each FT solution implements an active feedback loop to detect and repair faults quickly.

Driven by these three principles, we propose the general architecture of our FT dom0 in Figure 4.1. We interpret the latter as follows. dom0 is disaggregated in four unikernels, namely XenStore_uk, net_uk, disk_uk, and tool_uk. Some unikernels (e.g., device driver unikernels) are made of sub-components. We equip each unikernel and each sub-component with a feedback loop that includes fault detection (probes) and repair (actuators) agents. Both probes and actuators are implemented outside the target component.We associate our (local) dom0 FT solution with the (distributed) data center management system (e.g., OpenStack Nova) because the repair of some failures may require a global point of view. For instance, VM creation request's failure due to a lack of resources on the server may require to retry the request on another server. The data center management system can only take this decision. Therefore, each time a failure occurs, our system's first step repair solution is performed locally on the actual machine. Then, if necessary, a notification is sent to the data center management system.

A global feedback loop coordinates per-component feedback loops to handle concurrent failures. The latter requires a certain repair order. For instance, the failure of XenStore_uk is likely to cause the failure of other unikernels since XenStore acts as a storage backend for their configuration metadata. Therefore, XenStore_uk repair should be launched first, before the repair of the other unikernels. We implement the global feedback loop inside the hypervisor, which is the only component that we assume to be safe.

4.2.2 General fault model

This section presents generically the pVM (dom0) fault model that we target. Additional details are given in the next sections for each component. In the disaggregated architecture on which we build our FT solution, the dom0 components can be classified into two types : stateful (XenStore_uk) and stateless (net_uk, disk_uk, and tool_uk). We assume that all components may suffer from crash faults and that stateful components can

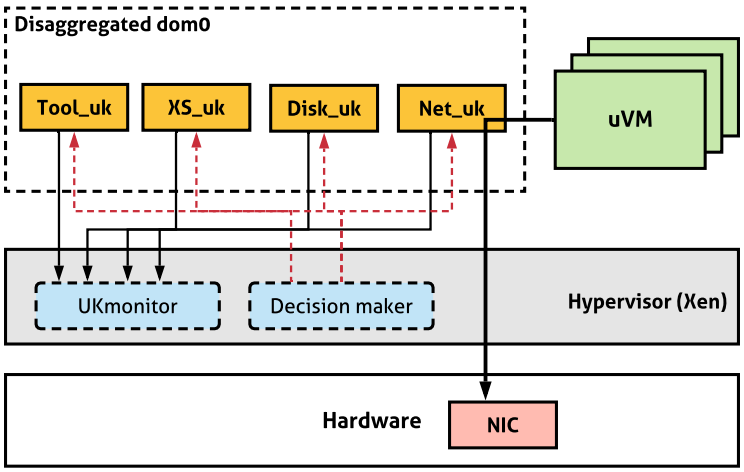

Figure 4.1 – Overall architecture of our FT pVM design.

also suffer from data corruption faults. Crash faults may happen in situations in which a component is abruptly terminated (e.g., due to invalid memory access) or hangs (e.g., due to a deadlock/livelock problem). These situations can make a component either unavailable, unreachable, or unresponsive when solicited. For stateful components, we are also interested in data corruption issues that may stem from various causes (e.g., an inconsistency introduced by a software crash, a sporadic bug, or hardware "bit rot"). Furthermore, our fault model encompasses situations in which several components are simultaneously in a failed state (either due to correlated/cascading failures) or due to independent issues. Besides, our work assumes that the code and data within the hypervisor component (i.e., *Xen*) are reliable or, more reasonably, that potential reliability issues within the hypervisor are addressed with state-of-the-art fault tolerance techniques such as ReHype [60] (discussed in §4.5). Our design requires small and localized modifications (318 LOCs) to the Xen hypervisor ; we believe that they do not introduce significant weaknesses in terms of reliability.

4.3 Implementation

To build our disaggregated dom0 architecture, we leverage the unikernels developed by the Xen project (Mini-OS and MirageOS). The motivation for these unikernels in the context of the Xen project is to contain the impact of faults in distinct pVM components. However, our contribution goes beyond the mere disaggregation of the pVM : we explain

how to support fine-grained fault detection and recovery for each pVM component. This section presents the implementation details of our fault tolerance (FT) solution for each dom0 unikernel, except (due to lack of space) for disk_uk, which is relatively similar to net_uk. For each unikernel, we first present the specific fault model that we target, followed by the FT solution (Table 4.3 presents a summary of the code size for each unikernel (existing code + modifications). Finally, the section presents the global feedback loop (which coordinates the recovery of multiple components) and discusses scheduling optimizations.

	net_uk	**disk_uk**	**tool_uk**	**xenstore_uk**
# Base LOCs	193*k*	350*k*	270*k*	8*k*
# Lines +	193	87	8	27

Table 4.3 – Lines of codes added to each unikernel codebase for fault tolerance.

4.3.1 XenStore_uk FT solution

XenStore is a critical metadata storage service on which other dom0 services rely. XenStore_uk runs within the MirageOS unikernel [65].

4.3.1.1 Fault model.

We consider two types of faults. The first type is *unavailability*, meaning that XenStore cannot handle incoming requests, due to bugs (hangs/crashes). The second type is silent *data corruption*; such issues may be caused by bit flips, defective hardware, or possibly malicious VMs (e.g., RowHammer attacks [19, 67]).

4.3.1.2 FT solution.

We use *state machine replication* and *sanity checks* to handle unavailability and data corruption, respectively. The overall architecture is depicted in Figure 4.2. Note that the memory footprint of a XenStore database is typically very small (lower than 1MB for 40 VMs).

Unavailability. We organize XenStore into several replicas (e.g., three in our default setup). Each replica runs in a dedicated unikernel based on MirageOS [65]. The set of

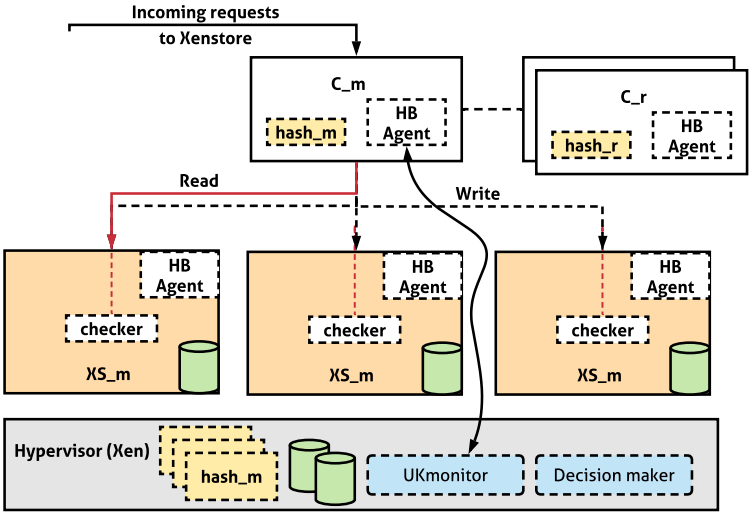

Figure 4.2 – Replication-based FT solution for XenStore.

replicas is managed by a coordinator running in a dedicated unikernel. Notice that the coordinator (noted C_m) is also replicated (the replicas are noted C_r) for FT. C_m chooses a XenStore replica to play the role of the master (noted XS_m). Let us note the other XenStore replicas XS_r. C_m is the XenStore entry point for requests sent by XenStore clients. We enforce this by modifying the `xs_talkv` function of the XenStore client library, used by the other components. C_m forwards read requests only to XS_m, while write requests are broadcast to all replicas.

We implement this *state machine replication* strategy using the *etcd* coordination system [27] deployed in a MirageOS unikernel. We choose etcd, because of its well-established robustness and its relatively lightweight resource requirements (compared to other coordination systems such as ZooKeeper [28]). Also, etcd has built-in support for high availability through strongly-consistent replication based on the Raft consensus algorithm [76]. In the rest of this section, we use the term etcd to refer to C_m and the C_r replicas.

We improve etcd to provide both failure detection and repair strategies, as follows. etcd is augmented with a heartbeat (HB) monitor for each XenStore replica. When a replica does not answer to a heartbeat, etcd pro-actively replaces the replica with a fresh version, whose state is obtained from another alive uncorrupted XenStore replica. This recovery process does not interrupt request handling by other replicas. In case of the unavailability of XS_m, etcd elects another master and forwards to it the in-progress requests that were assigned to the crashed master. C_m exchanges heartbeat messages with the hypervisor so that the latter can detect the simultaneous crashing of the former and the C_r replicas. In fact, the failure of one coordinator instance can be handled by the other instances without the hypervisor's intervention. The latter intervenes only when all instances crash at the same time.

Besides, we have modified the communication mechanism used between etcd and the other components. Instead of leveraging its default communication interface based on the HTTP protocol, we rely on virtual IRQs and shared memory. The motivation is twofold. First, this reduces the communication overheads on the critical path. Second, the utilization of HTTP would involve net_uk in the failure handling path of XenStore, thus adding a dependence of the latter with respect to the former. This dependency would make it challenging to handle cascading failures since net_uk already relies on XenStore.

To provide a highly available metadata storage service, an alternative design could consist of using the etcd instances as a complete replacement for the XenStore instances. This approach would reduce the number of unikernel instances and some communication steps between the pVM components. We have tried to implement this approach, but the achieved performance was significantly poorer : we observed request latencies that were higher by up to several orders of magnitude (microseconds vs. milliseconds). Indeed, XenStore and etcd are datastores with a fairly similar data model, but their implementations are optimized for different contexts (local machine interactions versus distributed systems). In addition, the design that we have chosen helps limit the modifications to be made to the implement the vanilla pVM components. In particular, this allows benefiting from new features and performance optimizations integrated into the vanilla XenStore codebase, e.g., regarding data branching and transactions.

Data corruption. This type of fault is handled by a *sanity check* approach implemented on all XenStore replicas, as described below. First, we make the following assumption : for a given logical piece of information that is replicated into several physical copies, we assume that there is at most one corrupted copy. Each etcd instance stores a hash of the content of the latest known uncorrupted XenStore database state. Besides, a sanity check agent (called *checker*) runs as a transparent proxy in each XenStore replica. Upon every write request sent to the XenStore service, each `checker` computes a new hash, which is forwarded to the etcd coordinator. If the hashes sent by all the replicas match, then this new value is used to update the hash stored by all the etcd instances. Upon receiving a read request, the master XenStore replica computes a hash of its current database state and compares it against the hash sent by the coordinator. If they do not match, a distributed recovery protocol is run between the etcd coordinators to determine if the corrupted hash stems from the coordinator or the XenStore master replica. In the former case, the hash of the coordinator is replaced by the correct value. In the latter case, the XenStore replica is considered faulty, and the etcd coordinator triggers the above-mentioned recovery process.

Total XenStore failure. In the worse case, all XenStore and/or etcd components can crash at the same time. In our solution, this situation is detected and handled by the hypervisor via the heartbeat mechanism mentioned above. The hypervisor relaunches the

impacted component according to a dependency graph (see §4.3.4). However, an additional issue is the need to retrieve the state of the XenStore database. To tolerate such a scenario without relying on the availability of the disk_uk (to retrieve a persistent copy of the database state), we rely on the hypervisor to store additional copies of the XenStore database and the corresponding hashes. More precisely, the hypervisor hosts an in-memory backup copy for the database and hash stored by each replica and each replica is in charge of updating its backup copy.

4.3.2 net_uk FT solution

Based on the Mini-OS unikernel [64], the net_uk component embeds the NIC driver. Following the *split driver model*, it proxies incoming and outgoing network I/O requests to/from user VMs. To this end, net_uk also runs a virtual driver called *netback* that interacts with a pseudo NIC driver called *netfront* inside the user VM. The interactions between netback and netfront correspond to a bidirectional producer-consumer pattern and are implemented via a ring buffer of shared memory pages and virtual IRQs. Overall, net_uk can be seen as a composite component encapsulating the NIC driver and the netback.

4.3.2.1 Fault model.

We are interested in mitigating the unavailability of net_uk. The latter can be caused by a crash of the NIC driver, the netback, or the whole unikernel. We assume that a fault in the NIC driver or the netback does not corrupt the kernel's low-level data structures. This is a viable assumption as we can run the NIC driver in an isolated environment similar to Nooks [90] or LXDs [69].

4.3.2.2 FT solution.

Our approach aims at detecting failures at two levels : a coarse-grained level when the whole unikernel fails and a fine-grained level for the NIC driver and netback failures. Before presenting the details of our solution, we first provide a brief background on the design of the I/O path, using the reception of a new packet as an example.

Once the NIC reports the arrival of a packet, a hardware interrupt is raised and trapped inside the hypervisor. The latter forwards the interrupt (as a virtual interrupt) to net_uk. The handler of that virtual interrupt is then scheduled inside net_uk. In general, the interrupt handler is organized in two parts namely *top half* (TH) and *bottom half* (BH). The

top half masks off interrupt generation on the NIC and generates a *softirq* whose handler is the bottom half. The latter registers a NAPI ("New API") function, aimed at polling for additional incoming network packets. The maximum number of packets that can be pooled using this mechanism is controlled via a *budget* and a *weight* parameter. Upon its completion, the bottom half unmasks interrupt generation by the NIC. Overall, this design allows limiting the overhead of network interrupts.

To handle NIC driver failures, we leverage the *shadow driver* approach introduced by Swift et al. [89]. The latter was proposed for bare-metal systems. We adapt it for a Xen virtualized environment as follows. The original *shadow driver* approach states that each (physical) driver to be made fault tolerant should be associated a with shadow driver, interposed between the former and the kernel. This way, a failure of the target driver can be masked by its shadow driver, which will mimic the former during the recovery period. The shadow driver can work in two modes : passive and active. In passive mode, it simply monitors the flow of incoming and completed requests between the kernel and the target driver. Upon failure of the target driver, the shadow driver switches to the active mode : it triggers the restart of the target driver (and intercepts the calls made to the kernel), and it buffers the incoming requests from the kernel to the target driver (which will be forwarded after the recovery process).

In our specific virtualized context, we do not create a shadow driver for each net_uk component. Instead, we consider an improved version of the netback driver as the shadow driver for both itself and the NIC driver (see Fig. 4.3.b). In this way, we reduce the number of shadow drivers and, as a consequence, the net_uk code base (complexity). When a bottom half handler is scheduled, a signal is sent to the hypervisor, which records the corresponding timestamp t_o. Once the execution of the bottom half ends, another signal is sent to the hypervisor to notify completion (see Fig. 4.3.a). If no completion signal is received by the hypervisor after $t_o + t_{max}$, where t_{max} is the estimated bottom half maximum completion time, the hypervisor considers that the NIC driver has failed, and triggers the recovery of the driver (using existing techniques [89]), as shown in Fig. 4.3.b. The tuning of t_{max} depends on budget and weight values (see above) and is empirically determined. In our testbed, the values used for budget and weight are 300 and 64 respectively, and t_{max} is about $4s$.

Regarding the failure of the netback, the hypervisor monitors the shared ring buffer producer and consumer counters between a netback and its corresponding frontend. If the netback's private ring counters remain stuck while the shared ring counters keep evolving, this lag is considered as a hint revealing the failure of the netback. Hence, the netback is reloaded (unregistered then registered). Meanwhile, its frontend's device attribute `otherend->state` value switches to `XenbusStateReconfigured` while the netback undergoes repair. Once the repair is complete, the latter value switches back to `XenbusStateConnected` and proceeds with the exchange of I/O requests with the netback.

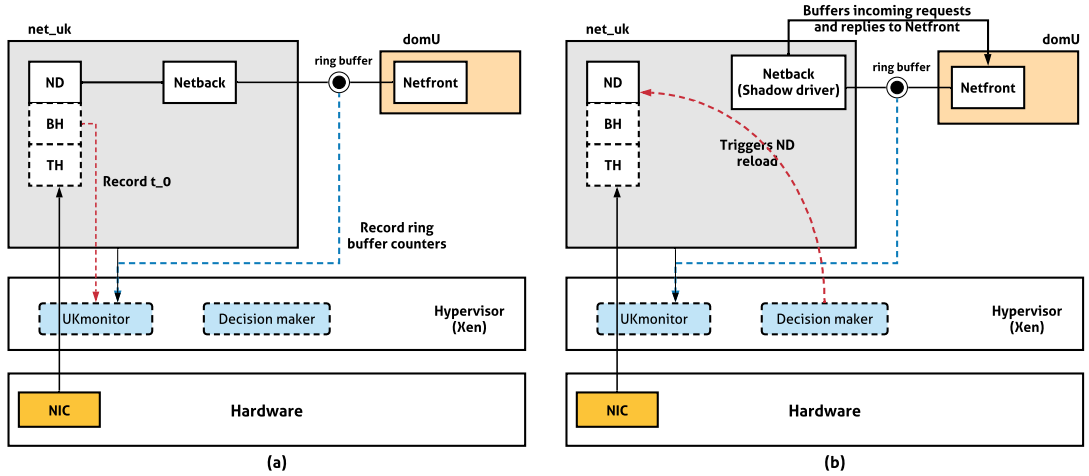

Figure 4.3 – net_uk, in which the shadow driver (netback) works either in passive *(a) or in* active *(b) mode. In the former mode (no NIC driver failure), the hypervisor records the bottom half handler (BH) starting timestamp t_o and awaits a completion signal before $t_o + t_{max}$, otherwise triggers NIC driver (ND) reload. In active mode (ND failure has been detected), the netback buffers requests and acks the netfront upon ND recovery.*

Regarding the failure of the entire unikernel, we adopt the same approach as TFD-Xen [41] : the hypervisor monitors the sum of the counters of the shared ring buffer used by all netbacks and their corresponding netfront drivers to detect a lag between the producer and the consumer counter. However, this approach alone cannot detect net_uk hanging when it is not used by any user VM. Therefore, we combine it with a heartbeat mechanism, also controlled by the hypervisor. A reboot of the net_uk VM is triggered when any of the two above-described detection techniques raises an alarm.

4.3.3 tool_uk FT solution

The tool_uk unikernel embeds the Xen toolstack for VM administration tasks (creation, migration, etc.). We use XSM (Xen Security Modules [104]) to introduce a new *role* (*tooldom*) which has fewer privileges than the original monolithic dom0 but enough for administrative services. It runs in an enriched version of Mini-OS [64], a very lightweight unikernel, part of the Xen project.

4.3.3.1 Fault model.

We strive to mitigate faults occurring during administrative operations. Apart from live migration (discussed below), the fault tolerance requirements for all the other administration tasks are already fully handled either locally, by the vanilla toolstack imple-

mentation, or globally, by the data center management system (e.g. Nova in OpenStack). In these cases, our solution provides nonetheless fast and reliable notifications regarding the failures of the local toolstack. We now describe the specific problem of resilient live migration. During the final phase of a live migration operation for a VM [1], the suspended state of the migrated VM is transferred to the destination host and upon reception on the latter, the VM is resumed. If a fault occurs during that phase, the migration process halts and leaves a corrupted state of the VM on the destination machine and a suspended VM on the sender machine.

4.3.3.2 FT solution.

We consider that a failure has occurred during the migration process if the sender machine does not receive (within a timeout interval) the acknowledgement message from the destination machine, which validates the end of the operation. As other unikernels in our solution, faults resulting in the crash/hang of the entire tool_uk are detected with a heartbeat mechanism and trigger the restart of the tool_uk instance. In both cases (partial or complete failure of the component), the repair operation for the failed migration is quite simple and consists in (*i*) first discarding the state of the suspended VM on the destination machine, (*ii*) destroying the VM on the destination machine, and (*iii*) resuming the original VM instance on the sender machine.

4.3.4 Global feedback loop

Our solution includes a global feedback loop for handling concurrent failures of multiple pVM components (and potentially all of them). Such a situation may or may not be due to a cascading failure. To handle such a situation in the most efficient way, the hypervisor embeds a graph thats indicates the dependencies between the different unikernels, which are represented in Figure 4.4. When a unikernel fails, the hypervisor starts the recovery process only when all unikernels used by the former are known to be healthy and reachable.

1. Xen adopts a pre-copy iterative strategy for live migration [18].

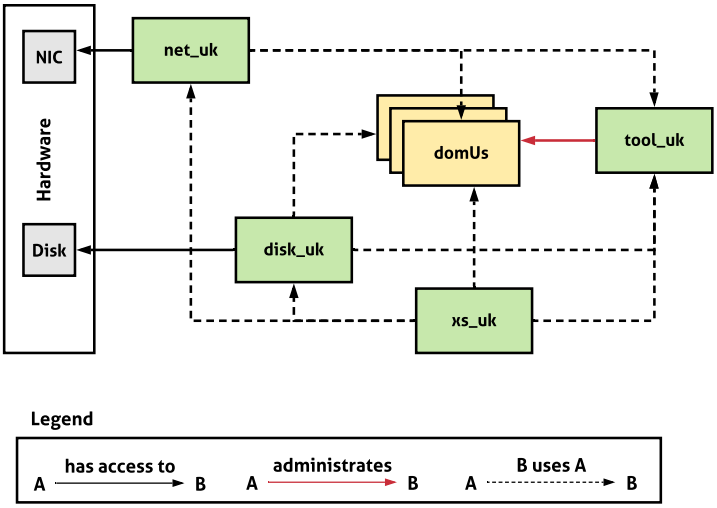

Figure 4.4 – Relationships between the different components of the disaggregated pVM.

4.3.5 Scheduling optimizations

The design that we have described so far, with the disaggregation of the pVM services into independent unikernel VMs and the usage of heartbeats to detect their failures, raises some challenges with respect to CPU scheduling. Indeed, it is non-trivial to ensure that these VMs are appropriately scheduled. On the one hand, due to the number of VMs resulting from the disaggregation, dedicating one (or several) distinct physical CPU core(s) to each unikernel VM would result in significant resource waste (overprovisioning). On the other hand, if such VMs are not scheduled frequently enough, they may not be able to send their heartbeats on time to the hypervisor (leading to false positives, and unneeded repair procedures), or, as a workaround, this may require to set longer timeouts (leading to slow detection of actual failures). In order to overcome the above-described issues, we slightly modify the CPU scheduler of the hypervisor. At creation time, each service VM is marked with a special flag and the hypervisor CPU scheduler guarantees that such VMs are frequently scheduled and sends a ping request to a unikernel VM before switching to it. Each service VM is granted a time slice of 5ms for heartbeat response. As an additional optimization, the scheduling algorithm is modified to skip the allocation of a CPU time slice to a unikernel VM if the latter has recently (in our setup, within the last 15ms) issued an “implicit” heartbeat (for example, in the case of the net_uk VM, a recent and successful interaction with the hypervisor for sending or receiving a packet is a form of implicit heartbeat). This avoids the cost of context switches to a unikernel VM solely for a ping-ack exchange when there are hints that this VM is alive.

4.4 Evaluation

This section presents the evaluation results of our prototype.

Evaluation methodology and goals. We evaluate both the robustness and the reactivity of our solution in fault situations. We first evaluate each dom0 service FT solution individually, meaning that a single failure (in a single component) is injected at a time in the system. Then, we consider the failure of several services at the same time. For each experiment, we consider a challenging situation in which both dom0 services and user VMs are highly solicited. Crash failures are emulated by killing the target component or unikernel. In order to simulate data corruption (in the case of XenStore_uk), we issue a request that overwrites a path (key-value pair) within the data store.

We are interested in the following metrics : (1) the overhead of our solution on the performance of dom0 services ; (2) the overhead of our solution on the performance of user VMs ; (3) the failure detection time ; (4) the failure recovery time ; (5) the impact of failures on dom0 services ; (6) the impact of failures on user VMs. The overhead evaluation is performed on fault-free situations. We compare our solution with vanilla Xen 4.12.1 (which provides almost no fault tolerance guarantees against pVM failures), Xoar [20] (periodic refresh), and TFD-Xen [41] (which only handles net_uk failures). For a meaningful comparison, we re-implemented the two previous systems in the (more recent) Xen version that we use for our solution. For Xoar, we use a component refresh period of 1 second, and the different components are refreshed sequentially (not simultaneously) in order to avoid pathologic behaviors.

Benchmarks. User VMs run applications from the TailBench benchmark suite [42]. The latter is composed of 8 latency-sensitive (I/O) applications that span a wide range of latency requirements and domains and a harness that implements a robust and statistically-sound load-testing methodology. It performs enough runs to achieve 95% confidence intervals $\leq 3\%$ on all runs. We use the client-server mode. The client and the server VMs run on distinct physical machines. The server VM is launched on the system under test. We left out 3 applications from the TailBench suite, namely *Shore, Silo* and *Specjbb*. Indeed, the two former are optimized to run on machines with solid state drives (whereas our testbed machine is equipped with hard disk drives), and *Specjbb* cannot run in client-server mode. In addition, we also measure the request throughput sustained by the Apache HTTP server (running in a user VM) with an input workload generated by the AB (ApacheBench) benchmark [9] (using 10,000 requests and a concurrency level of 10).

Testbed. All experiments are carried out on a 48-core PowerEdge R185 machine with AMD Opteron 6344 processors and 64 GB of memory. This is a four-socket NUMA machine, with 2 NUMA nodes per socket, 6 cores and 8 GB memory per NUMA node.

The dom0 components use two dedicated sockets and the user VMs are run on the two

other sockets. Providing dedicated resources to the pVM is in line with common practices used in production [68] in order to avoid interference. Besides, we choose to allocate a substantial amount of resources to the dom0 in order to evaluate more clearly the intrinsic overheads of our approach (rather than side effects of potential resource contention). We use Xen 4.10.0 and the dom0 runs Ubuntu 12.04.5 LTS with Linux kernel 5.0.8. The NIC is a Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet interface. The driver is bnx2. The machines are linked using a 1Gb/s Ethernet switch. Unless indicated otherwise, user VMs run Ubuntu 16.04 with Linux Kernel 5.0.8, configured with 16 virtual CPUs (vCPUs) and 16GB of memory. Concerning unikernels composing dom0, each is configured with 1 vCPU and 1 GB of memory (128MB for the XenStore instances). The real memory footprint during our evaluations is ≈500MB for every unikernel (≈100MB for each Xenstore instance). For fault-free runs, compared to vanilla Xen, we achieve 1-3% slowdown for I/O-intensive applications (disk or network). These results are similar to those reported with Xoar (original version [20] and our reimplementation) : the intrinsic performance overhead of disaggregation is low.

4.4.1 XenStore_uks

Recall that, in the fault model that we consider, XenStore is subject to both unavailability and data corruption faults. XenStore is highly solicited and plays a critical role during VM administration tasks. We use VM creation operations to evaluate XenStore, because this type of operation is one of the most latency-sensitive and also involves Xenstore the most (see Table 4.4).

VM create	VM destroy	VM migrate	vCPU hot-plug
53	47	24	12

Table 4.4 – Number of XenStore requests per type of VM administration operation.

4.4.1.1 Robustness

We launch a VM creation operation and inject a crash failure into the master XenStore replica (recall that we use a total of 3 XenStore instances) during the phase where XenStore is the most solicited.We repeat the experiment ten times and we report mean values. The observed results are as follows.

We observe that some VM creations fail with both vanilla Xen and Xoar. The latter, after the refresh period, is not able to replay the VM creation request because it has not been recorded. Besides, Xoar takes 1 second to detect the failure. Its recovery time is 22ms (a reboot of XenStore_uk). In contrast, using our solution, all VM creation operations complete successfully. Our solution takes 1.54ms and 5.04ms to detect crashing and

data corruption faults respectively. The recovery process for crashing and data corruption is 25.54ms (starting a new Xenstore_uk replica and synchronizing its database). The overall corresponding VM creation time is about 5.349s and 5.353s respectively for the two failure types, compared to 5.346s when no fault is injected.

4.4.1.2 Overhead

We sequentially execute ten VM creation operations (without faults). The mean VM creation time (until the full boot of the VM's kernel) for vanilla Xen, Xoar, and our solution (PpVMM) is respectively 4.445sec, 6.741sec, and 5.346sec. Our solution incurs about 20.27% ($\approx 900ms$) overhead. This is due to the fact that a VM creation operation generates mostly write requests (89% of the requests are writes), which require synchronization between all XenStore replicas. Read requests do not require synchronization. Fig. 4.5 reports mean, 95th- and 99th-percentile latencies for read and write requests, confirming the above analysis. The overhead incurred by our solution is significantly lower than the overhead of Xoar, which is about 51.65% ($\approx 2.3s$).

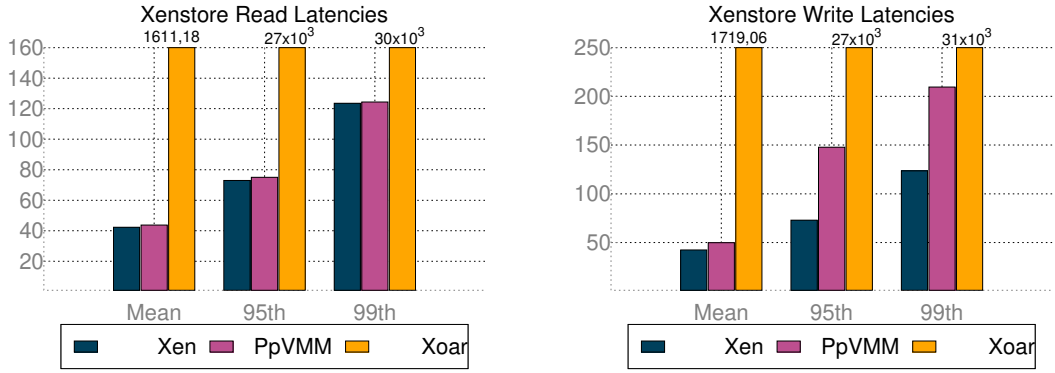

Figure 4.5 – Mean, 95th and 99th-percentile latencies of XenStore requests during 10 VM creation operations. The reported latencies are in µs.

4.4.2 net_uk

For these experiments, we run independently each TailBench application inside the user VM and we measure how it is impacted by crash failures.

4.4.2.1 Robustness

Recall that our solution enhances net_uk with several FT feedback loops in order to detect failures at different granularities : the unavailability of the subcomponents (NIC driver and netback) and the unavailability of the entire net_uk. Here, we only evaluate the robustness of our system facing NIC driver crashes because it allows us, through the same experiment, to compare fine-grained (FG) and coarse-grained (CG) FT solutions.

We inject a fault in the NIC driver at the middle of the execution of the benchmark. Table 4.5 and Table 4.6 present the results. We do not interpret Xoar results here (already discussed in §1.2.3). Besides, we do not show performance results for vanilla Xen because it is unable to achieve application completion in case of a net_uk failure.

We can see that the fine-grained solution allows quick detection compared to coarse-grained solutions (ours and TFD-Xen) : up to a 3.6x difference for detection and 1.4x for repair times (compared to our coarse-grained approach). TFD-Xen is faster to recover because it relies on net_uk replicas linked to backup physical NICs : instead of recovering a failed net_uk unikernel, it switches from one net_uk to another and reconfigures the bindings with the running user VMs. However, TFD-Xen requires at least $N+1$ physical NICs and $N+1$ net_uks to survive N net_uk faults, which results in resource waste and limited resilience over long time intervals. Furthermore, our fine-grained solution avoids packet losses, thanks to the use of a shadow driver that buffers packets in case of failure. For instance, we measured 212,506 buffered packets for the *sphinx* application. In contrast, the other solutions lead to broken TCP sessions (caused by packet losses) that occur during network reconfiguration (even for TFD-Xen, despite its short recovery time). Moreover, we can see that the fine-grained FT solution reduces the tail latency degradation compared to the coarse-grained solution. Considering the *sphinx* application for instance, the differences are respectively 24.88%, 12.88%, and 5.88% for the mean, 95th and 99th-percentile latencies.

Regarding the throughput measurements with the AB benchmark, we observe the following results. TFD-Xen achieves the best performance with 45 requests/s. The FG solution is relatively close with 42 requests/s (7.14% gap). In contrast, the CG approach is significantly less efficient with 29 requests/s (55.17%) and Xoar is much worse with 9 requests/s (400%).

	DT (ms)	**RT (s)**	**PL**
FG FT	27.27	4.7	0
CG FT	98.2	6.9	425,866
TFD-Xen [41]	102.1	0.8	2379
Xoar [20]	52×10^3	6.9	1,870,921

Table 4.5 – Robustness evaluation of different FT solutions for net_uk. The failed component is the NIC driver.
DT = Fault Detection Time ; RT = Fault Recovery Time ; PL = number of (outgoing) lost packets.

	sphinx			xapian			moses		
	mean	95th	99th	mean	95th	99th	mean	95th	99th
Xen	879.1	1696	1820.8	1.79	4.35	9.67	8.6	39.56	65.64
FG FT	1201.1	3100.1	3891.7	51.19	100.1	1700.3	73.2	473.2	1492.3
CG FT	1500.3	3499.4	4120.9	89.9	154.9	2101.4	100.5	591.5	1833.9
TFD-Xen	1159.2	2908.8	3304.2	50.1	98.1	1396.2	70.3	450.2	1101.4
Xoar	8100.4	11026.7	13590.3	5188.1	7238.9	8193.3	5120.4	5581.8	5909.3

	masstree			img-dnn		
	mean	95th	99th	mean	95th	99th
Xen	457.6	475.7	476.2	1.7	3.42	7.6
FG FT	821.1	1891.1	2122.8	88.2	440.2	1310.8
CG FT	1091.5	2099.1	2461.9	112.1	610.9	1503
TFD-Xen	788.3	1381.2	1631.7	80.2	398.3	1116.8
Xoar	10011.2	13444.5	140881.4	1491.9	4721.3	12390.4

Table 4.6 – Performance of TailBench applications during a net_uk failure (latencies in milliseconds). Lower is better.
The failed component is the NIC driver. The first line ("Xen") corresponds to a fault-free baseline.

4.4.2.2 Overhead

The experiment is the same as previous without fault injection. Table 4.7 presents the results. The overhead incurred by our solution is up to 12.4% for mean latencies, up to 17.3% for the 95th percentiles, and up 12.3% for the 99th percentiles. This overhead is due to periodic communication with the hypervisor to track the driver execution state (§4.3.2). Notice that TFD-Xen [41] incurs overhead up to 2.88% for mean latencies, 17.87% for the 95th percentiles, and up to 13.77% for the 99th percentiles.The overhead incurred by Xoar is much higher, as already discussed in §1.2.3.

Regarding the throughput measurements with the AB benchmark, we observe the following results compared to the vanilla Xen baseline (123 requests/s). Both TFD-Xen and our solutions (FG and CG) exhibit a noticeable but acceptable overhead (13.31%, 12%, and 15% respectively), whereas Xoar incurs a more pronounced performance degradation (1130% with a refresh period of 5s, Xoar still incurs a performance degradation of up to 697%, significantly worse than our approach).

	sphinx			xapian			moses		
	mean	95th	99th	mean	95th	99th	mean	95th	99th
Xen	879.11	1696	1820.84	1.79	4.35	9.67	8.6	39.567	65.642
FG FT	901.99	1711.11	1977.15	2.11	5.56	10.61	10.1	40.78	72.44
CG FT	900.12	1792.04	1963.5	2.12	5.96	11.05	11.9	41.3	72.12
TFD-Xen	889.91	1701.43	1911.33	2.11	4.98	10.89	9.12	40.44	73.19
Xoar	6616.44	9026.7	9590.54	3713.98	5535.77	5791.712	3019.33	3507.88	3660.65

	masstree			img-dnn		
	mean	95th	99th	mean	95th	99th
Xen	457.61	475.37	476.2	1.7	3.42	7.6
FG FT	460.2	491.58	494.3	1.92	4.2	8.21
CG FT	461.09	489.03	490.12	1.9	4.3	7.98
TFD-Xen	461.18	489.1	493.55	1.9	4.5	8.11
Xoar	8054.53	8642.85	8695	543.9	2526	9603

Table 4.7 – Performance of TailBench applications without net_uk failure (latencies in milliseconds). Lower is better.

4.4.2.3 CPU usage

Here, we are interested in CPU usage in the netdom. We compare the results obtained with Xen and PpVMM . We run in a guest VM, applications from the tailbench suite. Figure 4.6 reports CPU usage in the netdom for Xen(dom0) and PpVMM (netdom). We observe an overall 3% increase with PpVMM compared to vanilla Xen.

4.4.3 tool_uk

Contrary to other dom0 unikernels, tool_uk does not execute a task permanently.It only starts a task when invoked for performing a VM administration operation. The FT solution does not incur overhead when there are no failures. Therefore we only evaluate the robustness aspect. To this end, we consider the VM live migration operation because it is the most critical one. We run inside the user VM a Linux kernel compilation task and inject a failure during the second stage of the migration process, i.e., when a replica of the migrated VM has been launched on the destination machine, and the memory transfer is ongoing.

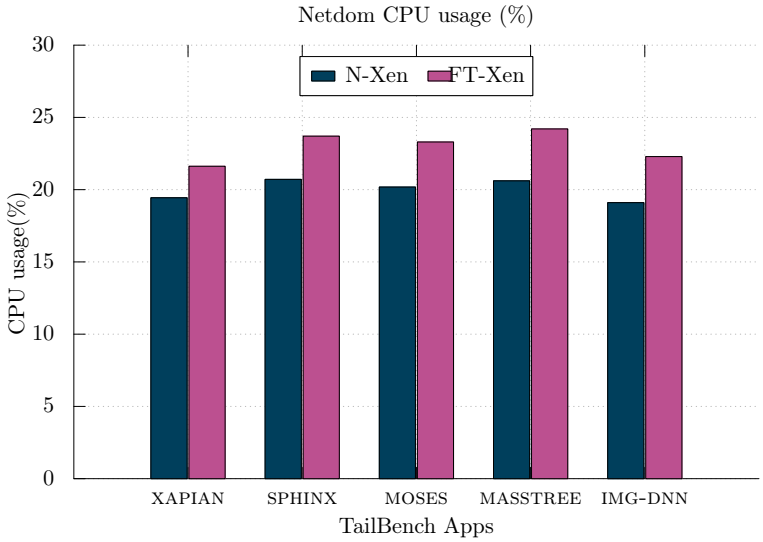

Figure 4.6 – Netdom CPU usage (Xen vs PpVMM). N-Xen stands for Normal Xen and FT-Xen stands for PpVMM.

We observe that vanilla Xen and Xoar lead the physical machine to an inconsistent state : the migration stops but both the original VM (on the source machine) and its replica (on the destination machine) keep running. This situation leads to resource waste because the replica VM consumes resources. Using our solution, the replica VM is stopped upon failure detection. The detection time is 800ms.

4.4.4 Global failure

We also evaluate the robustness of our solution when all the pVM components crash at the same time. We execute the sphinx application from TailBench in a guest and we inject faults to crash all the components simultaneously. In this case, the hypervisor detects the global crash and restores all unikernels in the appropriate order (see §4.3.4). The whole recovery of all unikernels takes 15.8s. Concerning application performance, we observe a downtime of 7.85s (corresponding to the time needed for XenStore_uk and net_uk to recover), but the application survives and finishes its execution correctly. We experience a huge degradation of tail latencies due to the long downtime but we allow full and transparent functional recovery of the user VM, unlike vanilla Xen, TFD-Xen, and with a much lower overhead than Xoar (esp. during failure-free execution phases).

4.4.5 Scheduling optimizations

We measure the benefits of our scheduling optimizations (§4.3.5) in terms of reactivity and CPU time used by our unikernels. Regarding reactivity, we run the *sphinx* application in a guest VM, and we trigger the crash of the net_uk. On average, with the scheduling optimizations, we detect the crash after 141.8ms compared to 149.5ms without, i.e., a

5.15% decrease. Besides, on a fault-free run, compared to a standard scheduling policy, the usage of implicit heartbeats allows a 13% decrease of the CPU time consumed by the pVM components.

4.5 Related work

4.5.1 pVM resilience.

The projects most closely related to our work are Xoar [20] and TFD-Xen [41]. Given that they are described in detail and evaluated in the previous sections.

Beyond Xoar, a number of projects have investigated the benefits of disaggregating the VMM into multiple isolated components. Murray et al. [66] modified the original Xen platform design in order to move the domain builder (a security-sensitive module within the Xen toolstack running in the pVM) to a separate virtual machine. This work did not investigate fine-grained disaggregation nor fault tolerance. Fraser et al. [29] revisited the design of the Xen platform in order to support "driver domains", i.e., the possibility to isolate each physical device driver in a separate VM. Our contribution builds on this work but also considers fine-grained fault-tolerance mechanisms within driver domains, as well as disaggregation and robustness of other pVM components.

As part of their Xen-based "resilient virtualization infrastructure" (RVI) [49, 51], Le and Tamir briefly discussed how to improve the fault tolerance of the pVM and the driver domains (dVMs). The failure of a driver domain is detected by agents within the dVM kernel and the hypervisor, which triggers the microreboot of the dVM. The failures of the services hosted by the pVM (e.g., XenStore) are detected by an agent running within the pVM. Upon such a detection, the agent issues a hypercall to the hypervisor, and the latter triggers a crash of the whole pVM. Hence, any failure of a given component hosted by the pVM (e.g., XenStore or toolstack) leads to downtime and full recovery for all the other components. In order to tolerate failures of the XenStore, its state is replicated in a dVM, and XenStore and VM management operations are made transactional, through the use of a log stored in a dVM. No detail is provided on the mechanisms used to enforce consistency and availability despite potential concurrent failures of the pVM-hosted components and the backup state in the dVM. Besides, the evaluation of this approach is focused on its resilience against synthetic fault injection. The authors of this solution do not provide any detailed performance measurements (with or without failures), and the code of the prototype is not available. Our work has similarities with this approach but, (i)

we apply fault tolerance techniques at a finer granularity, (ii) we explore the ramifications of the interdependencies between services, and (iii) we provide a detailed performance evaluation.

4.5.2 Hypervisor resilience.

Some works have focused on improving the resilience of the hypervisor. ReHype [49,50,60] is a Xen-based system, which leverages microreboot [13,25] techniques in order to recover from hypervisor failures, without stopping or resetting the state of the VMs (including the pVM and dVMs) or the underlying physical hardware. NiLyHype [115] is a variant of ReHype, which replaces the microreboot approach with an alternative component recovery technique named microreset (i.e., resetting a software component to a quiescent state that is likely to be valid rather than performing a full reboot) in order to improve the recovery latency. TinyChecker [16] uses nested virtualization techniques (more precisely, a small, trusted hypervisor running below a main, full-fledged hypervisor like Xen) in order to provide resilience against crashes and state corruption in the main hypervisor. Shi et al. [86] proposed a new modular, "deconstructed" design for Xen in order to thwart the most dangerous types of security attacks. Their work focuses on a redesign of the Xen hypervisor (not the dom0 pVM). All these works do not consider the services hosted in the pVM (or driver domains) and are mostly orthogonal to our work. Our contribution leverages these results (i.e., the fact that the hypervisor can be made highly resilient) in order to improve the fault tolerance of the pVM components.

The FTXen project [40] aims at hardening the Xen hypervisor layer so that it can withstand hardware failures on some "relaxed" (i.e., fast but unreliable) CPU cores. In contrast, our work considers the resilience of the pVM components on current hardware, with a fault model that is homogeneous/symmetric with respect to CPU cores.

Some recent projects aim at supporting live reboot and/or upgrade of VMMs without disrupting the VMs [26,114]. These techniques are focused on code updates for improving the safety and security of the hypervisor component. Hence, they are orthogonal to our contribution. This trend also highlights the crucial importance of our goal : improving the resilience of the remaining components, i.e., the pVM services.

4.6 Summary

Type-I hypervisors remain a key building block for cloud computing, and most of them are based on a pVM-based design. We have highlighted that, in this design, the pVM is a single point of failure. Besides, existing solutions only tackle a limited set of pVM services (device drivers) and/or require long failure detection/recovery times and

significant performance overheads. At the time of writing this dissertation, our contribution is the first to propose and demonstrate empirically, a complete approach allowing to achieve both high resilience (against failures of different components and concurrent failures of interdependent services) and low overhead. Our approach currently relies on manual tuning of some important parameters (e.g., for failure detection and scheduling) but, we envision that recently published works could help manage them in a more automated and robust way [101]. Another area for future work is the tuning and optimization of resource allocation for disagreggrated pVM components, which could be extended from the Closer principle we introduced in Chapter 2.

Conclusion

Contributions summary

In this dissertation, we showed how performance unpredictability could result from a component that does not spotlight the community. From resource sizing and placement on NUMA architectures to fault tolerance issues, Type-I hypervisors relying on a pVM must consider the latter issues. The design principle proposed regarding pVM resource sizing and placement allows dynamic management that scales with uVMs activities. Even though we didn't carry out stressful network evaluations, we think this is the right way to go (better than static approaches relying mostly on intuition). Regarding memory flipping side effects, we propose two hacks that permit control of what pages can become remote and enable cloud administrators to tweak the ideal strategy used based on their needs. Future work will characterize cloud workloads and define which strategy is better suited for each workload/behavior. However, we are conscious that memory flipping side effects are noticeable when long intensive network workloads run in uVMs, which may not be a common use case in the Cloud. Regarding the pVM fault tolerance, the results obtained with PpVMM are encouraging despite the manual configuration needed (on the heartbeat intervals, timeouts, etc.). However, autoconfiguration tools are more and more present and can alleviate the latter burden. Some of the approaches used, such as Xenstore replication design or netdom shadow driver, can be reused in use cases other than virtualization such as database integrity or improving I/O device drivers' resilience.

Short-term perspectives

Improve prototypes evaluation.

Virtualization is a large area, and there is always exciting stuff to work on. A prominent future work after going through this dissertation is to improve the evaluations. Ideally, we can test our contributions in production environments or large reserved clusters simulating real user workloads scenarios. This will highlight currently hidden pitfalls and strengthen the current designs.

Exploit hardware virtualization features.

Another work line is to investigate hardware features such as SR-IOV or Intel DDIO to examine if we can exploit them to improve the results we obtained while removing (or reducing) the interoperability limitations they impose. Ideally, we could have guest VMs devices exploiting SR-IOV to access I/O devices in a secure multiplexed manner while the pVM carries out monitoring and administrative tasks with dynamic resource provisioning and improved fault tolerance.

Auto-tuned parameters for pVM fault tolerance.

As mentioned earlier, our fault-tolerance approach relies on many manual parameters such as heartbeat intervals. We need to improve these either by providing tools that can help administrators compute them based on their needs or update our design for self-managed values.

Long-term perspectives

Reusing our contributions in FaaS environments.

Our contributions can be helpful in FaaS environments. To recall, FaaS is a model service where a user uploads a function on a FaaS platform and configures events that will trigger the latter function's execution. The user is billed based on the execution time and resources used by this function on each execution. Frameworks generally power faaS platforms with similar architecture as pVM-based hypervisors. Indeed, they generally have a component responsible for host administrative tools, schedule requests to compute nodes, and maintain databases used to query function states. Some of the problems encountered in pVM-based hypervisors also arise, such as resource provisioning, fault tolerance, and data locality. It can be interesting to show how far pVM-based hypervisors and FaaS platforms differ and see what can be reused.

Improving pVM-based hypervisors (VM vs containers ?)

The long battle VM vs containers is ongoing. However, improving pVM-based hypervisors to have startup times faster than containers may help till the balance in favor of VMs. However, the goal is not to show the superiority of VMs over containers but to find the

best in between design that benefits each advantage to have a more secure and fast virtualization base block. After investigating pVM-based hypervisors, the next step will be to investigate every container engine to highlight what can be fuse together.

References

[1] 451 RESEARCH, Last accessed 10/03/2020. https://www.prweb.com/releases/2017/11/prweb14952386.htm.

[2] ABE, Y., GEAMBASU, R., JOSHI, K., AND SATYANARAYANAN, M. Urgent virtual machine eviction with enlightened post-copy. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2016), VEE '16, Association for Computing Machinery, p. 51–64.

[3] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel virtualization technology for directed i/o. *Intel Technology Journal 10*, 03 (August 2006), 179–192.

[4] AHMAD, I., GULATI, A., AND MASHTIZADEH, A. Vic : Interrupt coalescing for virtual machine storage device io. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (USA, 2011), USENIXATC'11, USENIX Association, p. 4.

[5] AHMED, A., CONWAY, P., HUGHES, B., AND WEBER, F. Amd opteron shared memory mp systems.

[6] AL DANIAL, Last accessed 17/10/2020. https://github.com/AlDanial/cloc.

[7] AMD, Last accessed 10/05/2020. https://www.amd.com/en/technologies/virtualization-solutions.

[8] ANJALI, CARAZA-HARTER, T., AND SWIFT, M. M. Blending containers and virtual machines : A study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2020), VEE '20, Association for Computing Machinery, p. 101–113.

[9] APACHEBENCH, Last accessed 18/10/2020. http://tiny.cc/anu4nz.

[10] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. Linkbench : A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1185–1196.

[11] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev. 37*, 5 (Oct. 2003), 164–177.

[12] BUGNION, E., NIEH, J., AND TSAFRIR, D. *Hardware and Software Support for Virtualization*. Morgan and Claypool Publishers, 2017.

[13] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot — A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 3–3.

[14] CHAD PERRIN - MICROSOFT, Last accessed 10/04/2020. https://www.techrepublic.com/blog/it-security/the-danger-of-complexity-more-code-more-bugs/.

[15] CHAPMAN, M., AND HEISER, G. vnuma : A virtual shared-memory multiprocessor. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)* (San Diego, CA, June 2009), USENIX Association.

[16] CHENG TAN, YUBIN XIA, HAIBO CHEN, AND BINYU ZANG. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*.

[17] CHIANG, R. C., HUANG, H. H., WOOD, T., LIU, C., AND SPATSCHECK, O. Iorchestra : Supporting high-performance data-intensive applications in the cloud via collaborative virtualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 45 :1–45 :12.

[18] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation* (USA, 2005), NSDI'05, USENIX Association, p. 273–286.

[19] COJOCAR, L., RAZAVI, K., GIUFFRIDA, C., AND BOS, H. Exploiting Correcting Codes : On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P* (May 2019). Best Practical Paper Award, Pwnie Award Nomination for Most Innovative Research.

[20] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking up is hard to do : Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 189–202.

[21] CREASY, R. J. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development 25*, 5 (1981), 483–490.

[22] CYTHIA HARVEY, Last accessed on 10/01/2020. https://www.datamation.com/cloud-computing/cloud-adoption.html.

[23] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic management : A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 381–394.

[24] DELIMITROU, C., AND KOZYRAKIS, C. Hcloud : Resource-efficient provisioning in shared cloud systems. *SIGARCH Comput. Archit. News 44*, 2 (Mar. 2016), 473–488.

[25] DEPOUTOVITCH, A., AND STUMM, M. Otherworld : Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 181–194.

[26] DODDAMANI, S., SINHA, P., LU, H., CHENG, T.-H. K., BAGDI, H. H., AND GOPALAN, K. Fast and Live Hypervisor Replacement. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2019), VEE 2019, ACM, pp. 45–58.

[27] ETCD, Last accessed 19/10/2020. https://etcd.io.

[28] EXPLORING PERFORMANCE OF ETCD, ZOOKEEPER AND CONSUL CONSISTENT KEY-VALUE DATASTORES, Last accessed 13/10/2020. http://tiny.cc/8hu4nz.

[29] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe Hardware Access with the Xen Virtual Machine Monitor. In *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).

[30] GHAZAL, A., RABL, T., HU, M., RAAB, F., POESS, M., CROLOTTE, A., AND JACOBSEN, H.-A. Bigbench : Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1197–1208.

[31] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware* (Melbourne, Australia, 2006), Middleware '06, Springer-Verlag New York, Inc., pp. 342–362.

[32] GVISOR, Last accessed 10/10/2020. https://gvisor.dev/.

[33] HAN, J., AHN, J., KIM, C., KWON, Y., CHOI, Y.-R., AND HUH, J. The effect of multi-core on hpc applications in virtualized systems. In *Euro-Par 2010 Parallel Processing Workshops* (Berlin, Heidelberg, 2011), M. R. Guarracino, F. Vivien,

J. L. Träff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. Di Martino, and M. Alexander, Eds., Springer Berlin Heidelberg, pp. 615–623.

[34] HELWETT PACKARD. https://hewlettpackard.github.io/netperf/.

[35] HOSTING TRIBUNAL, 2020. https://hostingtribunal.com/blog/cloud-computing-statistics/.

[36] HUANG, Q., AND LEE, P. P. An experimental study of cascading performance interference in a virtualized environment. *SIGMETRICS Perform. Eval. Rev. 40*, 4 (Apr. 2013), 43–52.

[37] HYPERTRANSPORT CONSORTIUM, Last accessed on 11/10/2020. https://www.hypertransport.org.

[38] INTEL, Last accessed 09/05/2020. http://www.cs.columbia.edu/~cdall/candidacy/pdf/Uhlig2005.pdf.

[39] JIN, H., DENG, L., WU, S., SHI, X., AND PAN, X. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops* (2009), pp. 1–10.

[40] JIN, X., PARK, S., SHENG, T., CHEN, R., SHAN, Z., AND ZHOU, Y. FTXen : Making hypervisor resilient to hardware faults on relaxed cores. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2015), pp. 451–462.

[41] JO, H., KIM, H., JANG, J., LEE, J., AND MAENG, S. Transparent fault tolerance of device drivers for virtual machines. *IEEE Transactions on Computers 59*, 11 (Nov 2010), 1466–1479.

[42] KASTURE, H., AND SANCHEZ, D. Tailbench : a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)* (2016), pp. 1–10.

[43] KATACONTAINERS, Last accessed 10/09/2020. https://katacontainers.io/.

[44] KERNBENCH, Last accessed 14/10/2020. http://ck.kolivas.org/apps/kernbench/kernbench-0.50/.

[45] KNAUTH, T., AND FETZER, C. Dreamserver : Truly on-demand cloud services. In *Proceedings of International Conference on Systems and Storage* (New York, NY, USA, 2014), SYSTOR 2014, Association for Computing Machinery, p. 1–11.

[46] KNAUTH, T., KIRUVALE, P., HILTUNEN, M., AND FETZER, C. Sloth : Sdn-enabled activity-based virtual machine deployment. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2014), HotSDN '14, Association for Computing Machinery, p. 205–206.

[47] KOURAI, K., AND OOBA, H. Vmbeam : Zero-copy migration of virtual machines for virtual iaas clouds. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)* (2016), pp. 121–126.

[48] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock : Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, Association for Computing Machinery, p. 1–12.

[49] LE, M. Resilient Virtualized Systems. Ph.D. thesis 140007, UCLA Computer Science Department, March 2014.

[50] LE, M., AND TAMIR, Y. Applying Microreboot to System Software. In *2012 IEEE Sixth International Conference on Software Security and Reliability* (June 2012), pp. 11–20.

[51] LE, M., AND YUVAL, T. Resilient Virtualized Systems Using ReHype. Tech. Rep. 140019, UCLA Computer Science Department, October 2014.

[52] LEPERS, B., QUÉMA, V., AND FEDOROVA, A. Thread and memory placement on numa systems : Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 277–289.

[53] LINUX.DIE.NET. Irq balance, June 2013.

[54] LIU, M., AND LI, T. Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 325–336.

[55] MADHAVAPEDDY, A., LEONARD, T., SKJEGSTAD, M., GAZAGNAIRE, T., SHEETS, D., SCOTT, D., MORTIER, R., CHAUDHRY, A., SINGH, B., LUDLAM, J., CROWCROFT, J., AND LESLIE, I. Jitsu : Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (USA, 2015), NSDI'15, USENIX Association, p. 559–573.

[56] MAGENTO, Last accessed 14/10/2020. https://magento.com/.

[57] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 218–233.

[58] MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[59] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on*

Virtual Execution Environments (New York, NY, USA, 2005), VEE '05, Association for Computing Machinery, p. 13–23.

[60] MICHAEL LE, AND YUVAL TAMIR. ReHype : enabling VM survival across hypervisor failures. In *VEE '11 : Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2011), vol. 46, pp. 63–74.

[61] MICROSOFT, Last accessed 03/10/2020. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-.

[62] MICROSOFT, Last accessed 15/10/2020. https://cutt.ly/OgOozAI.

[63] MICROSOFT, Last accessed on 10/10/2020. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview.

[64] MINIOS, Last accessed 17/10/2020. https://github.com/mirage/mini-os.

[65] MIRAGE OS, Last accessed 10/10/2020. https://mirage.io.

[66] MURRAY, D. G., MILOS, G., AND HAND, S. Improving Xen Security Through Disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 151–160.

[67] MUTLU, O. The RowHammer Problem and Other Issues We May Face As Memory Becomes Denser. In *Proceedings of the Conference on Design, Automation & Test in Europe* (3001 Leuven, Belgium, Belgium, 2017), DATE '17, European Design and Automation Association, pp. 1116–1121.

[68] MVONDO, D., TEABE, B., TCHANA, A., HAGIMONT, D., AND DE PALMA, N. Closer : A New Design Principle for the Privileged Virtual Machine OS. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2019), pp. 49–60.

[69] NARAYANAN, V., BALASUBRAMANIAN, A., JACOBSEN, C., SPALL, S., BAUER, S., QUIGLEY, M., HUSSAIN, A., YOUNIS, A., SHEN, J., BHATTACHARYYA, M., AND BURTSEV, A. LXDs : Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 269–284.

[70] NATHAN, S., KULKARNI, P., AND BELLUR, U. Resource availability based performance benchmarking of virtual machine migrations. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2013), ICPE '13, Association for Computing Machinery, p. 387–398.

[71] NITU, V., OLIVIER, P., TCHANA, A., CHIBA, D., BARBALACE, A., HAGIMONT, D., AND RAVINDRAN, B. Swift birth and quick death : Enabling fast parallel guest boot and destruction in the xen hypervisor. *SIGPLAN Not. 52*, 7 (Apr. 2017), 1–14.

[72] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2009), CCGRID '09, IEEE Computer Society, pp. 124–131.

[73] NUTANIX, Last accessed 13/10/2020. https://www.nutanix.com/.

[74] OLIVIER, P., CHIBA, D., LANKES, S., MIN, C., AND RAVINDRAN, B. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2019), VEE 2019, Association for Computing Machinery, p. 59–73.

[75] ONGARO, D., COX, A. L., AND RIXNER, S. Scheduling i/o in virtual machine monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Seattle, WA, USA, 2008), VEE '08, ACM, pp. 1–10.

[76] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

[77] OPENNEBULA. Opennebula, July 2017.

[78] OPENSTACK.ORG. Openstack, Aug. 2017.

[79] ORACLE. Migration disabled with oracle vm server sr-iov, Last accessed 06/10/2020. https ://goo.gl/uneSHr.

[80] ORACLE, Last accessed on 01/10/2020. https://docs.oracle.com/cd/E50245_01/E50251/html/vmadm-config-server-memory.html.

[81] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, Association for Computing Machinery, p. 291–304.

[82] RANDAL, A. The ideal versus the real : Revisiting the history of virtual machines and containers. *ACM Comput. Surv. 53*, 1 (Feb. 2020).

[83] RAO, J., WANG, K., ZHOU, X., AND XU, C. Optimizing virtual machine scheduling in numa multicore systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (2013), pp. 306–317.

[84] REDHAT. Migration disabled with hyper-v sr-iov, Last accessed 09/10/2020. https ://goo.gl/qUtsQz.

[85] ROSENBLUM, M., AND WALDSPURGER, C. I/o virtualization : Decoupling a logical device from its physical implementation offers many compelling advantages. *Queue 9*, 11 (Nov. 2011), 30–39.

[86] SHI, L., WU, Y., XIA, Y., DAUTENHAHN, N., CHEN, H., ZANG, B., AND LI, J. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017* (2017).

[87] STEPHEN SPECTOR, Last accessed on 15/10/2020. http://www-archive.xenproject.org/files/Marketing/WhyXen.pdf.

[88] SVÄRD, P., HUDZIA, B., TORDSSON, J., AND ELMROTH, E. Evaluation of delta compression techniques for efficient live migration of large virtual machines. *SIGPLAN Not. 46*, 7 (Mar. 2011), 111–120.

[89] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering Device Drivers. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006), 333–360.

[90] SWIFT, M. M., MARTIN, S., LEVY, H. M., AND EGGERS, S. J. Nooks : An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (New York, NY, USA, 2002), EW 10, ACM, pp. 102–107.

[91] TEABE, B., TCHANA, A., AND HAGIMONT, D. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom, 2016), EuroSys '16, ACM, pp. 3 :1–3 :14.

[92] TEABE, B., TCHANA, A., AND HAGIMONT, D. Billing system CPU time on individual VM. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016* (2016), pp. 493–496.

[93] THOMADAKIS, M. The architecture of the nehalem processor and nehalem-ep smp platforms. *JFE Technical Report* (03 2011).

[94] ULRICH DREPPER - "WHAT EVERY PROGRAMMER SHOULD KNOW ABOUT MEMORY", Last accessed 14/10/2020. https://lwn.net/Articles/250967/.

[95] VIRTUALBOX, Last accessed on 10/10/2020. https://www.virtualbox.org/.

[96] VMWARE. Migration disabled with vmware vsphere sr-iov, Last accessed 07/10/2020. https ://goo.gl/d8MGD5.

[97] VMWARE, Last accessed 09/10/2020. https://www.vmware.com/products/workstation-pro.html.

[98] VMWARE, Last accessed 11/10/2020. https://www.vmware.com/products/esxi-and-esx.html.

[99] VMWARE, Last accessed on 10/10/2020. https://www.vmware.com/fr/products/fusion.html.

[100] VORON, G., THOMAS, G., QUÉMA, V., AND SENS, P. An interface to implement numa policies in the xen hypervisor. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, p. 453–467.

[101] WANG, S., LI, C., HOFFMANN, H., LU, S., SENTOSA, W., AND KISTIJANTORO, A. I. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, Association for Computing Machinery, p. 154–168.

[102] WOOD, T., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. Cloudnet : Dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2011), VEE '11, Association for Computing Machinery, p. 121–132.

[103] WORDPRESS, Last accessed 15/10/2020. https://wordpress.org/plugins/benchmark/.

[104] XEN SECURITY MODULES, Last accessed 16/10/2020. http://tiny.cc/zdu4nz.

[105] XEN.ORG. xen : free_domheap_pages : delay page scrub to idle loop., May 2014.

[106] XL, Last accessed 17/10/2020. https://wiki.xenproject.org/wiki/XL.

[107] XU, C., GAMAGE, S., LU, H., KOMPELLA, R., AND XU, D. vturbo : Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 243–254.

[108] XU, C., GAMAGE, S., RAO, P. N., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. vslicer : Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (Delft, The Netherlands, 2012), HPDC '12, ACM, pp. 3–14.

[109] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing : A gvisor case study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing* (USA, 2019), HotCloud'19, USENIX Association, p. 16.

[110] ZENG, L., WANG, Y., FENG, D., AND KENT, K. B. Xcollopts : A novel improvement of network virtualizations in xen for i/o-latency sensitive applications on multicores. *IEEE Transactions on Network and Service Management 12*, 2 (June 2015), 163–175.

[111] ZENG, L., WANG, Y., SHI, W., AND FENG, D. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *2013 International Conference on Cloud Computing and Big Data* (Dec 2013), pp. 267–274.

[112] ZHANG, I., DENNISTON, T., BASKAKOV, Y., AND GARTHWAITE, A. Optimizing VM checkpointing for restore performance in vmware esxi. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 1–12.

[113] ZHANG, I., GARTHWAITE, A., BASKAKOV, Y., AND BARR, K. C. Fast restore of checkpointed memory using working set estimation. *SIGPLAN Not. 46*, 7 (Mar. 2011), 87–98.

[114] ZHANG, X., ZHENG, X., WANG, Z., LI, Q., FU, J., ZHANG, Y., AND SHEN, Y. Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 93–105.

[115] ZHOU DIYU, AND TAMIR YUVAL. Fast Hypervisor Recovery Without Reboot. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (06 2018), pp. 115–126.

[116] ZHU, J., JIANG, Z., AND XIAO, Z. Twinkle : A fast resource provisioning mechanism for internet services. In *2011 Proceedings IEEE INFOCOM* (2011), pp. 802–810.

[117] ZIAKAS, D., BAUM, A., MADDOX, R. A., AND SAFRANEK, R. J. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects* (2010), pp. 1–6.

List of figures

1.1 Evolution of virtual machines and containers [82]. 8
1.2 General architecture of system-level virtualization against containers. . . 9
1.3 General architecture of Type-I hypervisors (bare-metal) against Type-II hypervisors (hosted). 11
1.4 Architecture of two widely used Type-I hypervisors, Xen and Hyper-V. Each has the notion of pVM to support the hypervisor with administrative and uVM I/O management tasks. 11
1.5 Example of a NUMA architecture. Nodes are linked via Interconnects (such as AMD HyperTransport [7] or Intel QuickPath Interconnect [38]). Local accesses (in green) are faster than remote accesses (in red). 12
1.6 A static resource allocation to the pVM can lead to its saturation, thus to performance unpredictability for user applications : (left) pVM's CPU load, (right) performance of the uVM which executes a (Wordpress) workload. The pVM has 2 CPUs for the top results and 12 CPUs for the bottom. 17
1.7 A static resource allocation to the pVM may lead to its saturation, thus to variable uVM (left) creation and (right) migration time. 18
1.8 uVM memory layout with memory flipping (a) and memory copy (b) - application performance for Big Bench (c). 20
1.9 Performance of the Stream benchmark : throughput (left axis / histogram boxes) and number of remote allocations (right axis / solid lines). 20
1.10 Mean and tail latencies for TailBench applications when self-refresh is enabled for the (disaggregated) pVM components. The results (lower is better) are normalized with respect to a baseline without self-refresh for the same metrics. 22

2.1 The new pVM architecture. 28
2.2 Packet reception workflow. 30
2.3 Multi-threaded migration process (lower is better). The uVM runs a memory intensive application. 36
2.4 Packet reception (left) and packet emission (right) improved by locality. (lower is better) 37

2.5 Packet reception improved by locality : multiple uVMs evaluation. Results are normalized over *Vanilla* (lower is better). 37
2.6 Disk operations (from *dd*) improved by locality. (lower is better). 38
2.7 (left) Macro-benchmarks improved by locality. The results are normalized over *Vanilla* (the first bar). (right) MC's CPU load. 39
2.8 Fairness of our solutions. (close to the first bar is better). 40
2.9 Our solution with all mechanisms, Kernbench results. The latter are normalized over *Vanilla* when the benchmark runs alone. For predictability, compare <alone>-<solution> and <col.>-<solution>. For performance improvement, lower is better. 41

3.1 Illustration of the dedicated page pool for memory flipping. 47
3.2 Illustration of the asynchronous memory migration. 49
3.3 uVM memory layout after running Big Bench : (a) for vanilla Xen, (b) for the page pool solution and (c) for the asynchronuous memory migration . 49
3.4 (a) Stream and (b) LinkBench benchmark results. 51

4.1 Overall architecture of our FT pVM design. 60
4.2 Replication-based FT solution for XenStore. 62
4.3 net_uk, in which the shadow driver (netback) works either in *passive* (a) or in *active* (b) mode. In the former mode (no NIC driver failure), the hypervisor records the bottom half handler (BH) starting timestamp t_o and awaits a completion signal before $t_o + t_{max}$, otherwise triggers NIC driver (ND) reload. In active mode (ND failure has been detected), the netback buffers requests and acks the netfront upon ND recovery. 66
4.4 Relationships between the different components of the disaggregated pVM. 68
4.5 Mean, 95th and 99th-percentile latencies of XenStore requests during 10 VM creation operations. The reported latencies are in μs. 71
4.6 Netdom CPU usage (Xen vs PpVMM). N-Xen stands for Normal Xen and FT-Xen stands for PpVMM. 75

List of tables

1.1 Differences between Type-I & Type-II hypervisors based on different parameters : performance, security, and hardware support. 10

2.1 The benchmarks we used for evaluations. 34
2.2 MC's needs for three cloud management systems. 35

3.1 Benchmark descriptions 50

4.1 Evolution of source code size for the Xen hypervisor and a Linux-based pVM. LOC stands for "Lines of Code". For Linux, we only consider `x86` and `arm` in the `arch` folder. Also, only `net` and `block` are considered in `drivers` folder (other folders are not relevant for server machines). We used Cloc [6] to compute the LOC. 57
4.2 Impact of the failure of the different dom0 services (`xl` tools, network/disk drivers, XenStore) on the VM management operations and on the applications (in user VMs). An *"A"* mark indicates that the failure *always* leads to unavailability while a *"S"* mark denotes correlated failures that occur *only in specific situations.* 58
4.3 Lines of codes added to each unikernel codebase for fault tolerance. . . . 61
4.4 Number of XenStore requests per type of VM administration operation. . 70
4.5 Robustness evaluation of different FT solutions for net_uk. The failed component is the NIC driver. DT = Fault Detection Time ; RT = Fault Recovery Time ; PL = number of (outgoing) lost packets. 72
4.6 Performance of TailBench applications during a net_uk failure (latencies in milliseconds). Lower is better. The failed component is the NIC driver. The first line ("Xen") corresponds to a fault-free baseline. 73
4.7 Performance of TailBench applications without net_uk failure (latencies in milliseconds). Lower is better. 74