Virtualization of micro-architectural components using software solutions

Vo Quoc Bao Bui

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

M. VO QUOC BAO BUI

le mardi 29 septembre 2020

Titre :

Virtualization of Micro-architectural Components Using Software Solutions

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. DANIEL HAGIMONT

M. ALAIN TCHANA

Rapporteurs :

M. JALIL BOUKHOBZA, UNIVERSITE DE BRETAGNE OCCIDENTALE

Mme FABIENNE BOYER, UNIVERSITE GRENOBLE ALPES

Membre(s) du jury :

M. FABRICE HUET, UNIVERSITE DE NICE SOPHIA ANTIPOLIS, Président

M. ALAIN TCHANA, ECOLE NORMALE SUP LYON ENS DE LYON, Membre

M. DANIEL HAGIMONT, TOULOUSE INP, Membre

To my family...

It's hardware that makes a machine fast.
It's software that makes a fast machine slow.
***Craig Bruce**.*

Acknowledgments

I would like to express my gratitude to Mr. Alain Tchana, Professor at École Normale Supérieure de Lyon and to Mr. Daniel Hagimont, Professor at Institut National Polytechnique de Toulouse for their supervision during the three years of my PhD. Their advices are very valuable and important to the work in this thesis. I also would like to thank all the jury members who spent their valuable time to evaluate my work.

I'm very thankful to my colleagues in SEPIA research team: Boris Teabe, Vlad Nitu, Kevin Jiokeng, Patrick Lavoisier Wapet, Djob Mvondo, Mathieu Bacou, Grégoire Todeschi, Tu Dinh, Stella Bitchebe,... Beside the helpful work-related discussions, their humor and welcome made me feel happy and joyful to be a part of the team.

I wish to express my thanks to The Embassy of France in Viet Nam for the scholarship during my three years in France.

Finally, I'm grateful to my family and friends who always support me and give me encouragements to pursuit my goals.

Abstract

Cloud computing has become a dominant computing paradigm in the information technology industry due to its flexibility and efficiency in resource sharing and management. The key technology that enables cloud computing is virtualization. Essential requirements in a virtualized system where several virtual machines (VMs) run on a same physical machine include performance isolation and predictability. To enforce these properties, the virtualization software (called the hypervisor) must find a way to divide physical resources (e.g., physical memory, processor time) of the system and allocate them to VMs with respect to the amount of virtual resources defined for each VM. However, modern hardware have complex architectures and some microarchitectural-level resources such as processor caches, memory controllers, interconnects cannot be divided and allocated to VMs. They are globally shared among all VMs which compete for their use, leading to contention. Therefore, performance isolation and predictability are compromised.

In this thesis, we propose software solutions for preventing unpredictability in performance due to micro-architectural components. The first contribution is called Kyoto, a solution to the cache contention issue, inspired by the polluters pay principle. A VM is said to pollute the cache if it provokes significant cache replacements which impact the performance of other VMs. Henceforth, using the Kyoto system, the provider can encourage cloud users to book pollution permits for their VMs.

The second contribution addresses the problem of efficiently virtualizing NUMA machines. The major challenge comes from the fact that the hypervisor regularly reconfigures the placement of a VM over the NUMA topology. However, neither guest operating systems (OSs) nor system runtime libraries (e.g., HotSpot) are designed to consider NUMA topology changes at runtime, leading end user applications to unpredictable performance. We presents eXtended Para-Virtualization (XPV), a new principle to efficiently virtualize a NUMA architecture. XPV consists in revisiting the interface between the hypervisor and the guest OS, and between the guest OS and system runtime libraries so that they can dynamically take into account NUMA topology changes.

Résumé

Le cloud computing est devenu un paradigme dominant dans l'industrie informatique en raison de sa flexibilité et de son efficacité dans le partage et la gestion des ressources. La technologie clé qui permet le cloud computing est la virtualisation. L'isolation et la prédictibilité de performance sont des exigences essentielles d'un système virtualisé où plusieurs machines virtuelles (MVs) s'exécutent sur une même machine physique. Pour mettre en œuvre ces propriétés, le logiciel de virtualisation (appelé l'hyperviseur) doit trouver un moyen de diviser les ressources physiques (par exemple, la mémoire physique, le temps processeur) du système et de les allouer aux MVs en fonction de la quantité de ressources virtuelles définies pour chaque MV. Cependant, les machines modernes ont des architectures complexes et certaines ressources de niveau micro-architectural telles que les caches de processeur, les contrôleurs de mémoire, les interconnexions ne peuvent pas être divisées et allouées aux MVs. Elles sont partagées globalement entre toutes les MVs qui rivalisent pour leur utilisation, ce qui conduit à la contention. Par conséquent, l'isolation et la prédictibilité de performance sont compromises.

Dans cette thèse, nous proposons des solutions logicielles pour prevenir la non-prédictibilité des performances due aux composants micro-architecturaux. La première contribution s'appelle Kyoto, une solution pour le problème de contention du cache, inspirée du principe pollueur-payeur. Une MV est pollueuse si elle provoque des remplacements importants de lignes de cache qui ont un impact sur la performance des autres MVs. Désormais, en utilisant le système Kyoto, le fournisseur peut encourager les utilisateurs du cloud à réserver des permis de pollution pour leurs MVs.

La deuxième contribution aborde le problème de la virtualisation efficace des machines NUMA. Le défi majeur vient du fait que l'hyperviseur reconfigure régulièrement le placement d'une MV sur la topologie NUMA. Cependant, ni les systèmes d'exploitation (OSs) invités ni les librairies de l'environnement d'exécution (par exemple, HotSpot) ne sont conçus pour prendre en compte les changements de topologie NUMA pendant leur exécution, conduisant les applications de l'utilisateur final à des performances imprévisibles. Nous présentons eXtended Para-Virtualization (XPV), un nouveau principe pour virtualiser efficacement une architecture NUMA. XPV consiste à revisiter l'interface entre l'hyperviseur et l'OS invité, et entre l'OS invité et les librairies de l'environnement d'exécution afin qu'ils puissent prendre en compte de manière dynamique les changements de topologie NUMA.

Contents

1 Introduction **1**

2 Background **4**

2.1 Virtualization . . . 5
2.1.1 Definitions . . . 5
2.1.2 CPU Virtualization . . . 7
2.1.3 Memory Virtualization . . . 11
2.1.4 I/O Virtualization . . . 14
2.2 Processor Caches . . . 18
2.2.1 Definitions . . . 18
2.2.2 Operating principle . . . 20
2.2.3 Associativity . . . 21
2.3 Non-Uniform Memory Access (NUMA) . . . 22
2.3.1 The hardware view . . . 22
2.3.2 The Linux kernel view . . . 23
2.4 Virtualization of Micro-architectural Components . . . 24
2.5 Synthesis . . . 28

3 Kyoto: Taxing Virtual Machines for Cache Usage **29**

3.1 Motivations . . . 30
3.1.1 Problem statement . . . 30
3.1.2 Problem assessment . . . 31
3.1.2.1 Experimental environment . . . 31
3.1.2.2 Benchmarks . . . 31
3.1.2.3 Metrics . . . 32
3.1.2.4 Evaluation scenarios . . . 32
3.1.2.5 Evaluation results . . . 33
3.2 The Kyoto Principle . . . 34
3.2.1 Basic idea: “polluters pay” . . . 35
3.2.2 The Kyoto’s scheduler within Xen . . . 36
3.2.3 Computation of llc_cap_{act} . . . 37
3.3 Evaluations . . . 38
3.3.1 The processor is a good lever . . . 38

3.3.2 Equation 3.1 vs LLC misses (LLCM): which indicator as the *llc_cap*? . . . 39
3.3.3 KS4Xen's effectiveness . . . 40
3.3.4 Comparison with existing systems . . . 42
3.3.5 Kyoto's overhead . . . 43
3.4 Discussion . . . 46
3.5 Related Work . . . 47
3.6 Synthesis . . . 49

4 When eXtended Para–Virtualization (XPV) Meets NUMA 50
4.1 Motivations . . . 51
4.1.1 Description . . . 52
4.1.2 Limitations . . . 52
4.1.3 Synthesis . . . 56
4.1.4 Hot-(un)plug as a solution? . . . 56
4.2 eXtended Para-Virtualization . . . 57
4.2.1 Principle . . . 57
4.2.2 Methodology for making legacy systems XPV aware . . . 58
4.3 Technical Integration . . . 61
4.3.1 Xen modifications . . . 62
4.3.2 Linux modifications . . . 63
4.3.3 Application-level modifications: the HotSpot Java virtual machine use case . . . 64
4.4 Evaluations . . . 65
4.4.1 Experimental setup . . . 65
4.4.2 XPV implementation efficiency . . . 66
4.4.3 vNUMA vs blackbox solutions . . . 68
4.4.4 XPV facing topology changes . . . 70
4.4.4.1 XPV facing topology changes caused by vCPU loadbalancing . . . 71
4.4.4.2 XPV facing topology changes caused by memory ballooning . . . 72
4.4.5 Automatic NUMA Balancing (ANB) limitations . . . 73
4.4.6 XPV internals . . . 74
4.5 Related Work . . . 75
4.5.1 Industrial solutions . . . 75
4.5.2 Academic solutions . . . 76
4.5.3 Positioning of our work . . . 76
4.6 Synthesis . . . 77

5 Conclusion and Perspectives **78**
5.1 Conclusion 78
5.2 Perspectives 79

Bibliography **81**

Chapter 1

Introduction

Nowadays, many organizations tend to outsource the management of their physical infrastructure to hosting centers. By this way, companies aim at reducing their cost by paying only for what they really need. This trend, commonly called cloud computing, is general and concerns all field of Information Technology. Notably, recent years have seen HPC application developers and industries thinking about the migration of their applications to the cloud [81].

In this context, the majority of platforms implements the Infrastructure as a Service (IaaS) cloud model where customers buy virtual machines (VM) with a set of reserved resources. One of the essential properties of virtualization is that it provides isolation among VMs running on the same physical machine. Isolation takes different forms, including security (sandboxing) and performance. Regarding security, isolation between VMs means that operating systems (and their applications) running in VMs are executing in separate address spaces and are therefore protected against illegal (bogus or malicious) accesses from other VMs. Regarding performance, isolation means that the performance of applications in one VM should not be influenced or depend on the behavior of other VMs running on the same physical machine.

Performance isolation and predictability are hard to achieve in a virtualized context due to complex architectures of modern hardware. Some micro-architectural components such as last level cache (LLC) can't be properly and/or efficiently divided and virtualized, resulting in performance interference caused by contention. In addition, machines have evolved to NUMA multicore architectures where there is a complex interconnect to connect NUMA nodes, each contains a memory bank and several cores. The performance promises

of NUMA architectures is nearly achieved in bare-metal systems with heuristics that place the memory and the threads of the processes on the nodes [75, 53, 52, 56]. However, in a virtualized system, the guest OS cannot implement such heuristics as the hypervisors can blindly change the NUMA topology of the guest VMs in order to balance the workload in the underlying hardware.

In this thesis, we propose two contributions as follows:

Kyoto: A software solution to the issue of LLC contention. This solution is inspired by the polluters pay principle. A VM is said to pollute a cache if it provokes significant cache replacements which impact the performance of other VMs. We rely on hardware counters to monitor the cache activity of each VM and to measure each VM cache pollution level. A VM which exceeds its permitted pollution at runtime has its CPU capacity reduced accordingly.

eXtended Para-Virtualization (XPV): A new principle to efficiently virtualize a NUMA architecture. XPV consists in revisiting the interface between a hypervisor and a guest OS and between the guest OS and the system runtime libraries (SRLs) in order to dynamically adapt NUMA policies used in the guest OS and the SRLs when the NUMA topology of the VM changes. By doing so, XPV allows each layer in the virtualization stack to implement what it does best: optimization of resource utilization for the hypervisor and NUMA resource placement for the guest OS and the SRLs.

Publications that constitute this thesis:

1. Alain Tchana, Bao Bui, Boris Teabe, Vlad Nitu, Daniel Hagimont. Mitigating performance unpredictability in the IaaS using the Kyoto principle. Middleware 2016: 17th ACM/IFIP/USENIX International Middleware Conference, Dec 2016, Trento, Italy. pp. 1-10.

2. Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, et al.. When eXtended Para-Virtualization (XPV) meets NUMA. EUROSYS 2019: 14th European Conference on Computer Systems, Mar 2019, Dresde, Germany. pp.7.

Other publications:

1. Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. Everything You Should Know About Intel SGX Performance on Virtualized Systems. Proc. ACM Meas. Anal. Comput. Syst. 3, 1, Article 5 (March 2019), 21 pages. DOI:https://doi.org/10.1145/3322205.3311076.

The rest of this thesis is organized as follows. Chapter 2 presents the background of our work including virtualization, processor cache and NUMA and discusses the virtualization of several micro-architectural components. Chapter 3 and 4 present in detail our two contributions: Kyoto and XPV respectively, their implementation as well as their evaluation. Finally, chapter 5 concludes our work and discuss the future works.

Chapter 2

Background

Contents

2.1 Virtualization **5**
2.1.1 Definitions 5
2.1.2 CPU Virtualization 7
2.1.3 Memory Virtualization 11
2.1.4 I/O Virtualization 14
2.2 Processor Caches **18**
2.2.1 Definitions 18
2.2.2 Operating principle 20
2.2.3 Associativity 21
2.3 Non-Uniform Memory Access (NUMA) **22**
2.3.1 The hardware view 22
2.3.2 The Linux kernel view 23
2.4 Virtualization of Micro-architectural Components **24**
2.5 Synthesis **28**

We present here the background of this thesis: virtualization and virtualization techniques in section 2.1, processor caches in section 2.2, NUMA architecture in section 2.3 and an overview of virtualization solutions for different micro-architectural components in section 2.4. The two sections 2.2 and 2.3 cover micro-architectural components which are the sources of the performance unpredictability we address in this work.

2.1 Virtualization

2.1.1 Definitions

Virtualization isn't a new concept. It finds its root back to the late 1960s. Back then, it had been seen as a cost-effective technique for "organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications" [57]. A popular use case was to allow several developers to share an expensive computer system. However, by the 1980s, with the appearance of personal computers and the maturity of operating systems (OSs), virtualization didn't gain much attention from computer system researchers. When hardware becomes more powerful over time, interests in virtualization across academics and industry have been raised again. While virtualization is still regarded as an effective and flexible way for sharing computing resources, it can help address some limitations of the current system architectures, such as security, reliability and OS migration. It quickly becomes a driving force of innovation in both software and hardware industries. Several improvements have been made in OSs, systems management and hardware designs to account for better virtualization. A new computing paradigm called *cloud computing* where computing resources are provided to users on demand has been widely adopted thanks to the flexibility of virtualization technology.

Virtualization refers to the process of creating a virtual representation of something, such as CPU, memory, network card, disk storage. A combination of virtualized CPUs, memory and I/O devices that forms a complete and isolated computing environment is defined as a *virtual machine* (VM). The latter is created and managed by a system software called *hypervisor* or *virtual machine monitor* (VMM) and can run its own independent OS instance, called *the guest OS*. However, not any piece of software that creates and manages VMs is considered as hypervisor. According to Popek and Goldberg [88], a hypervisor has three essential characteristics: equivalence, performance and safety.

- **Equivalence:** A program executing in a virtual environment created by the hypervisor performs identically to its execution on hardware. The differences in executions caused by timing dependencies and resource availability are acceptable.
- **Performance:** A "statically significant" amount of program instructions must be executed directly by the hardware without the intervention from

the hypervisor. As a result, at worst, the virtual environment only shows minor decreases in speed.

- **Safety:** The hypervisor is in complete control of system resources. It means (1) a VM is limited to only using the amount of resources allocated to it and (2) it's possible for the hypervisor to regain control of resources already allocated. In other words, VMs are required to be isolated from each other as well as from the hypervisor.

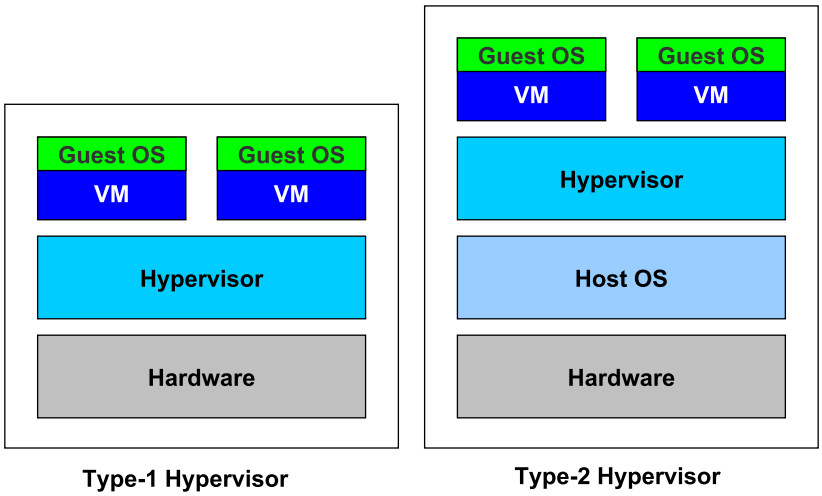

Figure 2.1: Two types of hypervisors.

Several hypervisors have been developed throughout the years and are usually (and loosely) classified into two categories depending on whether they run on a bare machine or on a host OS (Fig. 2.1):

- **Type-1:** The hypervisor runs on a bare machine and directly controls system resources. The emphasis here is on system scheduling and resource allocation [36]. The hypervisor itself makes the resource allocation and scheduling decisions. XEN [32], VMWare ESX Server [17] and Microsoft Hyper-V are examples of type-1 hypervisors.

- **Type-2:** The hypervisor runs on a *host OS*. It's the latter that plays the role of resource allocator and scheduler of the system. The hypervisor and VMs appear as normal processes to the host OS. Examples of type-2

hypervisors are KVM [70], Oracle VirtualBox [105], VMWare Workstation [35]. Although the performance of type-2 hypervisors may not be as good as that of type-1 hypervisors, their implementation is less complex.

2.1.2 CPU Virtualization

The requirements defined by Popek and Goldberg have been served as a guideline for developing hypervisors. Popek and Goldberg went further and proposed a theorem which determines if a given instruction set architecture (ISA) is virtualizable, in the sense that any OS being able to run on the hardware can be run inside a VM without modifications [88]: "*For any conventional third-generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions*". An instruction is considered *sensitive* if it can update the machine state or its semantics depend on the machine state. On the other hand, a *privileged* instruction is one that can only be executed in supervisor mode and will cause a trap when executed in user mode. The virtualization approach implied by the theorem is **direct execution** with **trap-and-emulate** which consists of, as its name suggests, (1) directly executing non-sensitive instructions on the processors; (2) trapping and then emulating sensitive instructions. In this way, the VMM can remain in control and configure the virtualized hardware correctly to achieve virtualization goals.

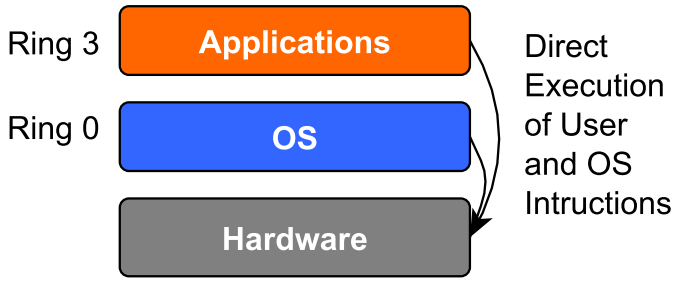

Figure 2.2: x86 architecture in a non-virtualized environment [8].

As stated above, researches on virtualization went inactive for some time. ISAs developed during that time, such as MIPS, x86 and ARM, didn't take virtualization into account. The x86 architecture, for example, contains several problematic sensitive but unprivileged instructions (pushf, popf, iret,...) [93].

As a result, it is not virtualizable according to the Popek and Goldberg theorem. However, there are other techniques, namely **binary translation** and **paravirtualization**, that can satisfy the virtualization requirements of equivalence, performance and safety. Later, when virtualization became more popular, hardware producers incorporated **hardware assisted** features into their processors helping the x86 architecture conform to the theorem.

The x86 hardware. Before going into detail about different virtualization techniques, it's worth noting that the x86 hardware is assumed by default in this thesis. As shown in the Fig. 2.2, in the x86 architecture, there are four levels of privilege, namely Ring 0, 1, 2 and 3. Ring 0 is the most privileged level, Ring 3 is the least. In a non-virtualized environment, Ring 0 is used for the OS, applications run at Ring 3 while Ring 1 and 2 are usually unused.

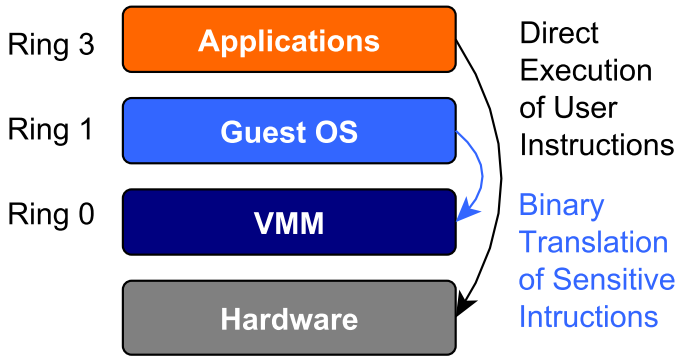

Figure 2.3: Full virtualization on x86 hardware [8].

Binary translation. This technique is usually employed to run binary programs compiled for a specific ISA on processors with a different ISA [97]. The way it works is to read and analyze the source instructions, translate them into the new instructions that are executable on the target hardware and execute them. The process can be done statically or dynamically.

Dynamic binary translation can be applied for virtualization to run unmodified OSs on x86 architecture. The idea is to translate sensitive but unprivileged instructions to safer ones that provide the intended effects on the VM. To speed up performance, translated code is cached. Optimization techniques such as "block chaining", which allows blocks of translated code to be executed one after another without transferring control back to the hypervisor, can be used. However, using binary translation for user-level programs could still introduce up to 5x slowdown [94]. To address the performance issue, VMware's solution

is to combine direct execution with binary translation: direct execution used to run user applications, binary translation used to run the guest OS [35]. This solution (Fig. 2.3) is referred to as **full virtualization**.

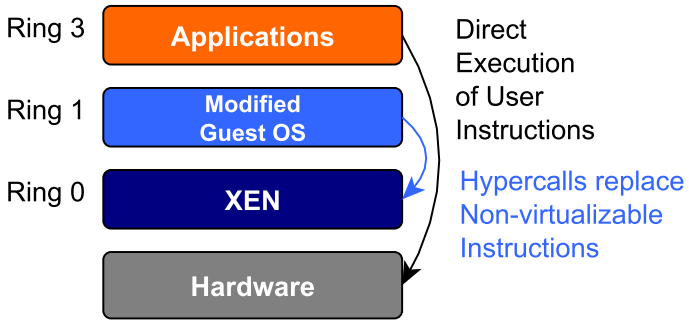

Figure 2.4: XEN paravirtualization on x86 hardware [8].

Paravirtualization. While binary translation adapts problematic instructions at the binary level, paravirtualization does that at the source code level. Therefore, this technique can only be used for open source OSs, such as Linux, FreeBSD. Although the application scope is limited, paravirtualization gives better performance than binary translation.

XEN [32] is a popular open source type-1 hypervisor that uses paravirtualization. To provide virtualization support for the x86 hardware, the OS's code for x86 architecture needs to be updated in a way that non-virtualizable instructions are unused. These instructions are replaced by hypercalls, which are calls from the guest OS to the hypervisor. The latter can then take appropriate actions to properly emulate the VM state. With these changes, all instructions to be executed are now virtualizable. XEN can use direct execution with trap-and-emulate to implement virtualization (Fig. 2.4).

Hardware assisted virtualization. To overcome the lack of virtualization support from the hardware, hypervisor developers had to find a way to adapt the guest OS at either source code or binary level. This made hypervisors more complex than they should be. As the need for virtualization became inevitable, Intel and AMD both have updated their products to add virtualization extensions. While Intel introduced VT-x, AMD developed AMD-V. Both are virtualization technologies for x86 platforms. The main goal is to eliminate the need for paravirtualization and binary translation. This helps hypervisors become simpler, more robust but still maintain a high level of performance and supports for unmodified guest OSs. We focus our discussion on Intel VT-x. Al-

though Intel VT-x and AMD-V are not entirely the same, they rely on similar principles.

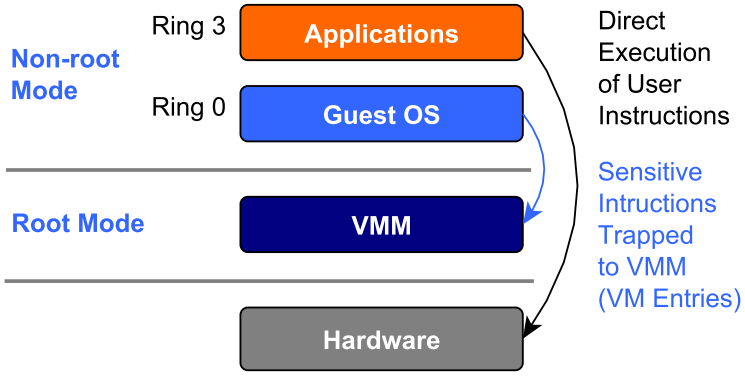

Figure 2.5: Hardware assisted virtualization on x86 (Intel) hardware [8].

VT-x enabled processors have two modes of operation: root operation and non-root operation [83]. Both operating modes offer four privilege levels. In general, root operation is intended for use by a hypervisor, while a guest OS runs in non-root operation. Transitions between the two modes are allowed. Precisely, there are two kinds of transition: a transition from root operation to non-root operation is called a *VM Entry*, and a transition from non-root operation to root operation is called a *VM Exit*. Upon a VM Entry, the so-called host processor state is stored and the guest processor state is loaded. Similarly, a VM Exit causes the guest state to be saved and the host state to be loaded. The two processor states are parts of a hardware-based data structure named the virtual machine control structure (VMCS).

Processors behave differently depending on which mode of operation they are in. In non-root mode, the processor behavior is limited and modified to facilitate virtualization. For instance, sensitive instructions and events now cause VM Exits which give control back to the hypervisor. It's also possible for system developers to configure conditions triggering VM Exits via the VMCS. With these architectural changes, Intel VT-x makes the x86 hardware virtualizable according to the Popek and Goldberg's criteria (Fig. 2.5).

2.1.3 Memory Virtualization

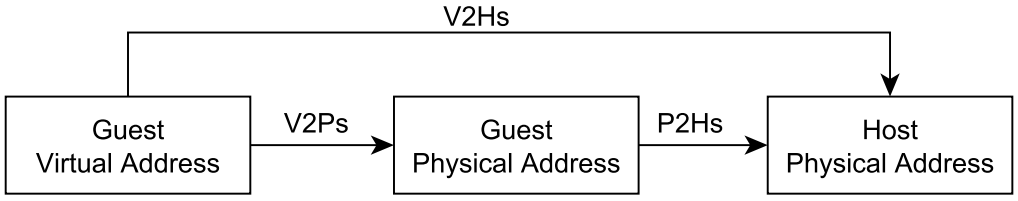

Figure 2.6: An overview of address translation in a virtualized environment.

Virtualization adds a new layer into the software stack alongside with a new address space. In a virtualized environment, there are basically three address spaces: guest virtual address (GVA), guest physical address (GPA) and host physical address (HPA). GVAs are what is exposed to applications by the guest OS. HPAs are real machine addresses used by the hardware to fetch and store the data. GPAs are addresses assumed by guest OSs as main memory but this memory is virtualized.

The main objective is to translate a GVA to a HPA. Address translations are usually done via a technique called paging, which consists of dividing memory into pages and using page tables (PTs) mapping pages between two address spaces. The guest OS sets up a set of virtual-to-physical page tables (V2Ps) for each process. The hypervisor *can* maintain two following sets of PTs: physical-to-host page tables (P2Hs) and virtual-to-host page tables (V2Hs). All these PTs are not necessarily needed at the same time, a virtualization technique can mainly rely on one or two of them. The actual address translation is however carried out by the PT walker, which is a special hardware of the memory management unit (MMU). Its job is to walk through the PTs of choice (pointed to by the *CR3* register) to get the desired address. Fig. 2.6 shows a high level overview regarding how the address translation can be done. We present here three well-known techniques: **shadow paging**, **direct paging** and **hardware assisted paging**.

Shadow paging. The guest OS is unaware of the existence of the hypervisor. In this case, the hypervisor maintains the V2Hs in the shadow. These PTs are usually referred to as the shadow page tables (SPTs). It is only the SPTs that are used by the PT walker to perform the address translation. Therefore, the performance of address translation is close to that of a native, non-

virtualized system. However, there are some sources of overhead in shadow paging (Fig. 2.7). First, the hypervisor must intervene when the guest OS updates its PTs in order to reflect the changes to the SPTs accordingly. This is usually done by a technique called *memory tracing*, which allows the hypervisor to mark any page in guest physical memory as read-only. Any attempt to write to a read-only page will result in a page fault, transferring control to the hypervisor. Second, the PT walker may update the accessed and dirty bits in the SPTs, these changes need to be reflected back in the guest. Third, when the guest tries to schedule a new process, it updates the guest CR3 register to establish the new process's PTs, the hypervisor must intercept this operation, set the shadow CR3 value pointing to the corresponding SPTs. As a result, if the VM causes too many context switches or PT updates, its performance may suffer greatly.

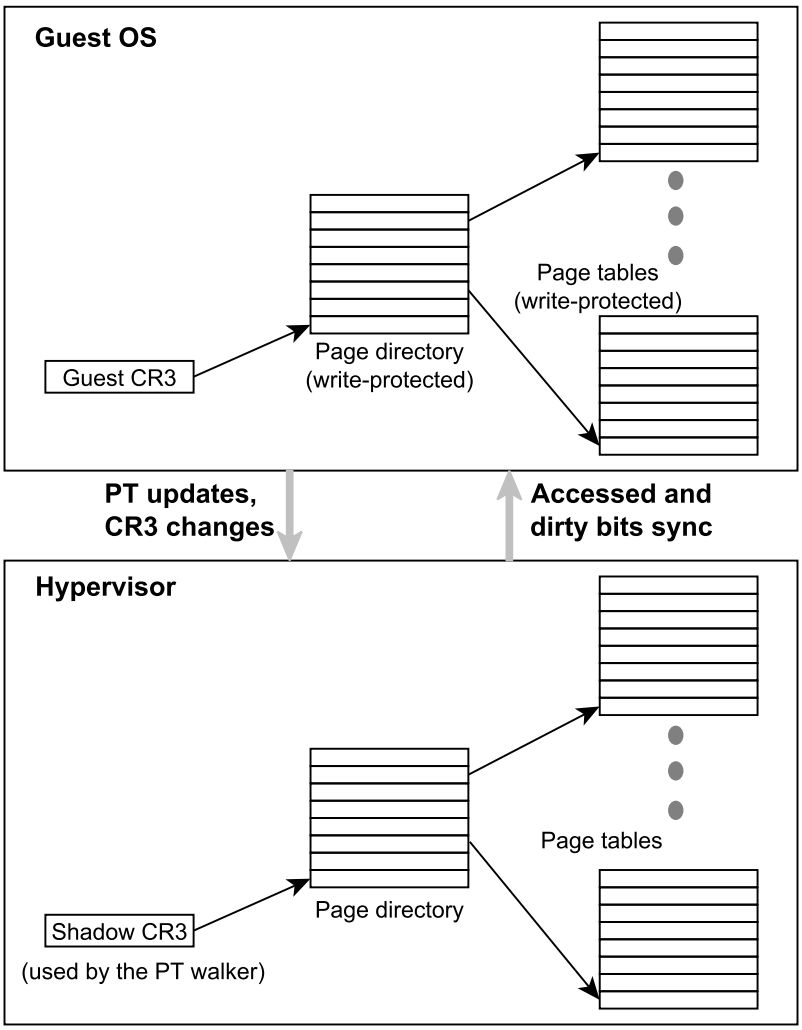

Figure 2.7: Shadow paging with 2-level PTs [15].

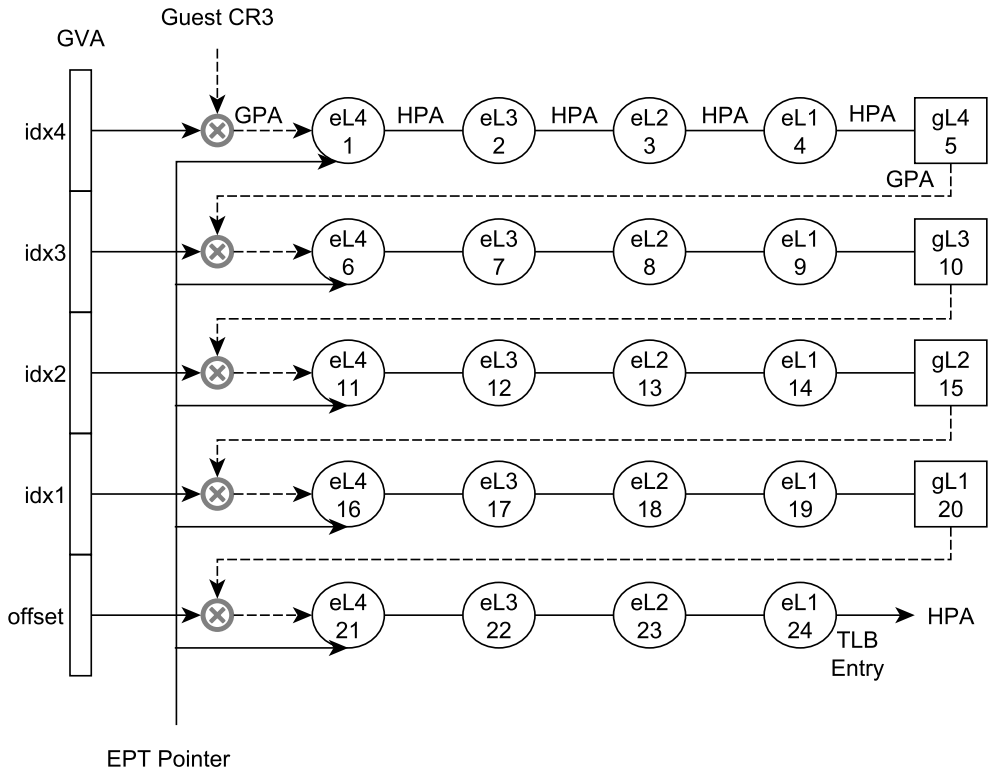

Figure 2.8: Address translation with EPT (4-level PTs; eLi is EPT level i; gLi is guest PT level i).

Direct paging. This technique is used by XEN to provide memory virtualization for paravirtualized VMs. As the guest OS knows that it runs in a virtualized environment, the hypervisor lets it manage the V2Hs directly (hence the name direct paging). The guest OS has controlled access to necessary information in order to correctly maintain the V2Hs. The hypervisor is still there to validate any update to the PTs made by the guest OS. As the latter has no write permission to the PTs, the changes have to be applied via hypercalls to the hypervisor.

Hardware assisted paging. Modern Intel's hardware has implemented a virtualization extension called Extended Page Tables (EPT) or Nested Page Tables (NPT) in case of AMD's. EPT adds to the VMCS the EPT pointer field that points to the P2Hs. The guest OS maintains the V2Ps which are pointed to by the guest CR3 register. This means the two PTs are now exposed to the hardware. The PT walker can perform a so-called 2-dimensional walk using both PTs to get a HPA from a GVA. Precisely, after each pass through

the V2Ps to translate a GVA to a GPA, the P2Hs are then used to derive a HPA from the GPA. As a result, EPT helps eliminate calls or traps into the hypervisor in case of the PT update synchronization. However, the cost of an address translation with EPT becomes more expensive as both PTs are used. For example, supposing both PTs have 4 levels in their structure, EPT requires 24 memory references in total as opposed to only 4 memory references in a native page walk (Fig. 2.8).

2.1.4 I/O Virtualization

I/O devices are important components for a computer system. Fortunately, just like in case of processor and memory, even with the lack of hardware supports for virtualization, they can still be virtualized purely by software techniques with the costs of engineering time and performance. However, for high performance use cases, hardware virtualization extensions are required.

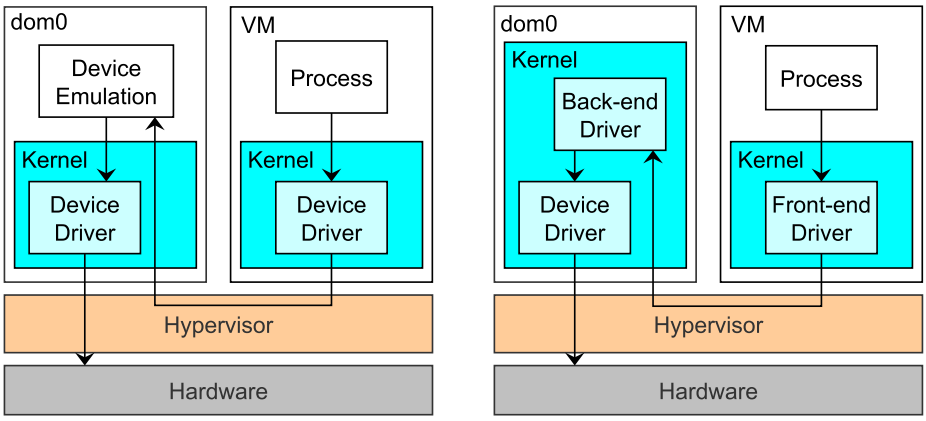

Figure 2.9: Architectures for software-based I/O virtualization (simplified): emulation (left) and paravirtualization (right). The hypervisor routes I/O requests from the guests to a privileged VM, denoted "dom0", having device drivers installed to perform the back-end of I/O operations.

I/O emulation (full virtualization). A traditional approach for I/O virtualization consists of exposing virtual I/O devices to the guests while implementing the semantics of physical devices in a device emulation layer. In general, the OS can interact with I/O devices via the following mechanisms: Port-mapped IO (PIO), Memory-mapped IO (MMIO), Direct Memory Access (DMA) and interrupts. With proper configuration, guest's PIOs and MMIOs

can be trapped, interrupts can be injected into the guest. Emulating DMA is easy for the hypervisor as it can read from and write to the guest's memory pages. All of these emulations can be done with or without VT-x/AMD-V support. To fulfill I/O requests from the guests, the hypervisor has two options: using device drivers embedded within itself (VMware ESXi) or relying on a privileged guest equipped with necessary device drivers (XEN, shown in Fig. 2.9). While using embedded drivers in the hypervisor may help reduce latency, the second option has the advantages of simplicity and portability.

I/O paravirtualization. Although I/O emulation provides good compatibility, it usually has high performance overheads. This is due to the fact that the physical devices that are emulated are not designed to support virtualization. For example, there are multiple register accesses involved in sending or receiving a single Ethernet frame using the Intel 82540EM Gigabit Ethernet Controller [36]. This is inefficient in a virtualized setup as the matching emulation code (e.g., e1000 in QEMU) would cause a lot of VM exits per frame sent or received as well.

It takes time for hardware makers to keep up with the virtualization train. However, it's possible to emulate a virtual device whose specification is efficient for virtualization (i.e., to minimize the number of VM exits involved). The guest OS and the hypervisor have to agree to work on such a device for I/O operations. The device is said to be paravirtualized and the guest needs to install a paravirtualized driver to work with it. This driver, which is also known as the front-end driver, is the first in the two components in the architecture of I/O paravirtualization. The second component is the back-end driver that runs in the hypervisor (or in a privileged guest) and serves as the underlying implementation for the device (Fig. 2.9).

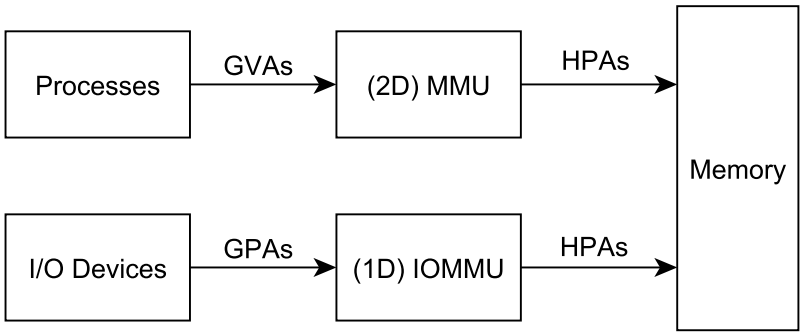

Figure 2.10: Address translation with 2D MMU for processes and 1D IOMMU for I/O devices [36].

Direct device assignment. Instead of exposing a virtual I/O device, the hypervisor can directly assign the physical device to the VM for exclusive usage. This solution may provide the best performance for an I/O-intensive VM as the latter can interact with the device with minimal or no involvement from the hypervisor. However, apart from some obvious drawbacks like the lack of scalability and portability, there are concerns about safety as well, especially for DMA capable devices. The fundamental problem is the mismatch of the notion of physical addresses used by the guest OS and the device. The guest OS directs the device using the GPAs while the device expects to work with the HPAs for I/O operations. This means the device could potentially access to memory belonging to the hypervisor or other VMs. A software-based solution for this is to modify the device driver to do the address translation. However, exposing the HPAs to the driver doesn't eliminate the safety issues. Additional hardware support is necessary for a secure and safe direct device assignment.

Many chip vendors have updated their hardware to include the I/O memory management unit (IOMMU), such as Intel's Virtualization Technology for Directed I/O (VT-d) [21], AMD's I/O Virtualization Technology (AMD-Vi) [24]. IOMMU has a component called *DMA remapping hardware* that can do the address translation using the page tables mapping the virtual addresses used by the device (e.g., the GPAs) to the HPAs. Basically, the way the DMA remapping hardware works for I/O devices is similar to how the MMU works for processes (Fig. 2.10). However, unlike regular memory accesses, I/O devices don't expect page faults for DMA operations. For that reason and because the hypervisor doesn't know exactly which pages are used by the device, when the hypervisor assigns a device to a guest, it usually pins the entire guest's memory.

I/O device sharing. Direct device assignment can achieve a native performance for guest's I/O operations but is not a scalable solution. Each direct assigned device can only be used by one VM at a time and the number of devices that are plugged into the host system is limited (comparing to the number of guests). To make direct device assignment scalable, the Single Root Input/Output Virtualization or SR-IOV was proposed. It is an extension for PCI Express (PCIe) devices. Traditionally, it's the responsibility of the hypervisor to multiplex I/O devices for the VMs. In case of a SR-IOV device, the resource sharing function is built in at the hardware level. However, some minimal support is still required from the system software in order for the SR-IOV to work as a resource sharing mechanism.

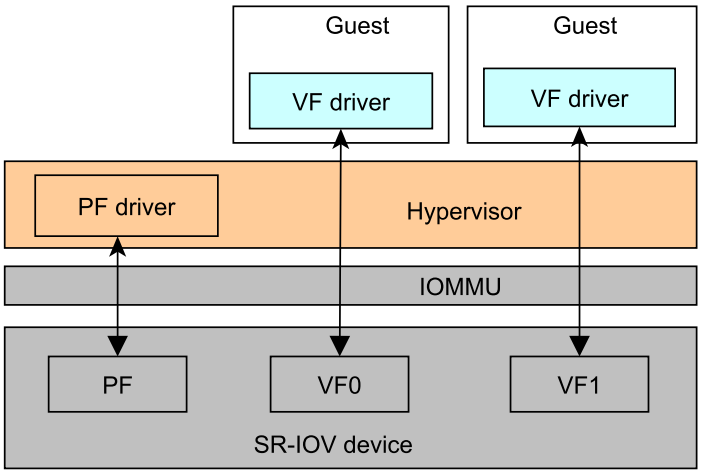

Figure 2.11: Direct device assignment with SR-IOV.

A SR-IOV device works by presenting at least one physical function (PF) and multiple virtual functions (VFs) [16]. A PF acts as a full-featured PCIe function with the ability to configure and manage the SR-IOV functionality (e.g., allocate, deallocate and configure the VFs). A VF is a simple PCIe function that can process I/O but with a limited configuration space. Each VF can be assigned to a VM as a separate device instance. In Fig. 2.11, the hypervisor uses the PF driver to configure the VFs of a SR-IOV device (e.g., a SR-IOV NIC) and assigns each VF for a guest. The VF's configuration space is presented to the guest as the device's configuration space. The assigned VF is recognized and usable thanks to the VF driver residing in the guest.

Supposing a VF of a SR-IOV NIC is assigned to the guest. The latter can receive an Ethernet packet as follows [19]. When the Ethernet packet arrives at the NIC, the Layer 2 sorter, which is configured by the PF driver, puts the packet into the dedicated receive queue of the target VF. The packet is then moved to the memory space of the guest via DMA with the target memory location configured by the VF driver. After the DMA operation is completed, the NIC fires an interrupt which is handled by the hypervisor. The latter generates a virtual interrupt to inform the guest that the packet has arrived. For any operation that has global effect, the VF driver must communicate with the PF driver. It's up to the vendor to implement the communication path between the two drivers.

2.2 Processor Caches

2.2.1 Definitions

The processor caches are layers of memory located between the main memory and a processor. One of the main activities of the processor is to access data (read or write) in memory. However, the speed of a computer's main memory is much slower than the processor's. Therefore, when the CPU tries to access memory, it has to wait for a period time during which it does not execute instructions. To minimize this waiting time, a cache memory is inserted between the CPU and the main memory. This mechanism allows the processor to access a very fast cache memory before accessing the main memory.

Cache memories are part of the so-called memory hierarchy. Fig. 2.12 shows a hierarchy of different types of memory available in a computer system. As we go up in this hierarchy, the cost and speed increase while the size decreases. At the highest level, there are the registers that are used for computations. The cache memories are actually placed between the registers and the main memory. In fact, with this structure, we can consider that for each level of memory in the hierarchy, the smaller and faster storage unit at level n serves as a cache for level n + 1 below.

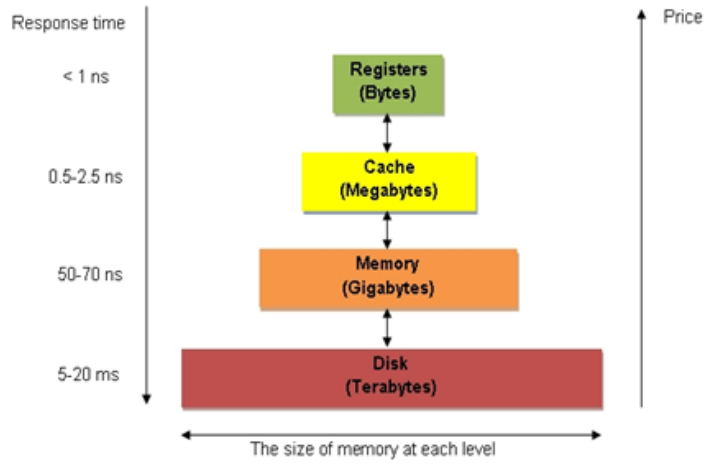

Figure 2.12: Memory hierarchy inside a computer system.

It is possible to integrate several levels of processor cache in a computer system. Today's machines often have three levels of cache memory (namely L1, L2 and L3, Fig. 2.13). The relationship between two levels of cache can

be inclusive or exclusive. For example, the L2 cache is said to be inclusive if it is forced to contain all the data inside the L1 cache. In case of an exclusive cache, the data can be in the L1 cache or in the L2 cache but never in both. When there is a cache fault at a cache level, the data is searched in the next cache level. A cache fault in the last level cache (LLC) triggers the loading of data from memory. The accessed data is replicated in all caches (except for exclusive caches). Moreover, today machines are multicore, potentially multi sockets, there are multiple local caches. As the shared data can be cached in different places, the problem of cache coherence where multiple local copies of the data need to be in-sync with each other arises. Snoopy or directory-based protocols are commonly used by the hardware to ensure coherency.

The memory hierarchy works effectively because of the principles of data locality. There are two forms of locality that many programs tend to follow. When a piece of data is accessed for the first time, it is very likely to be revisited later. So it's worth keeping it in the cache. This is the principle of temporal locality. Programs also tend to use the data in the memory area located near the recently accessed data. Therefore, prefetching this memory area in the cache could speeds up the performance. This is the principle of spatial locality. For example, the instructions of a program are executed one after the other and the next instruction to be executed is often placed in the location immediately after the current instruction (except for the branch instructions).

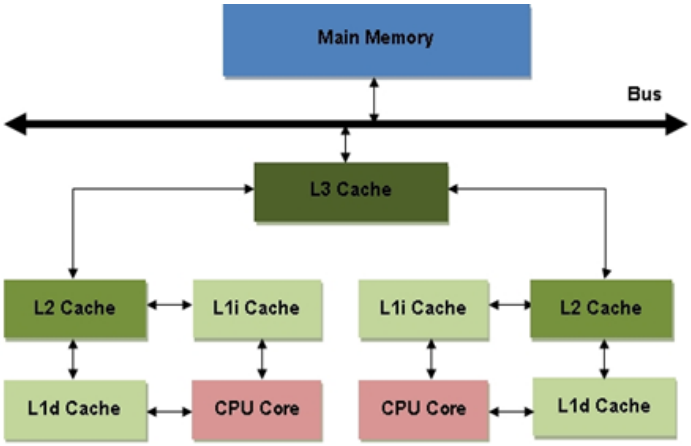

Figure 2.13: Example of hardware with three levels of memory cache.

2.2.2 Operating principle

The main memory is partitioned into contiguous data blocks. The cache memory is also partitioned into blocks of the same size as those in the main memory, but in fewer numbers. These blocks of memory are called cache lines. The size of each cache line can vary from 64 to 256 bytes. A cache line represents the smallest unit of data that can be transferred between the cache and main memory. That is, it is impossible to load only a portion of a cache line from memory. On the other hand, data words in a cache line can be accessed individually by the processor.

The processor does not directly access the cache. In fact, the cache memories are not addressable and invisible from the processor's point of view. When the processor requests a read or write at a location in the memory, this access is intercepted by the cache. If the requested data is present in the cache, there is a cache hit. Otherwise, if it isn't found, we have a cache miss or a cache fault. A cache miss is potentially much more expensive compared to a cache hit as the data may need to be loaded from the main memory.

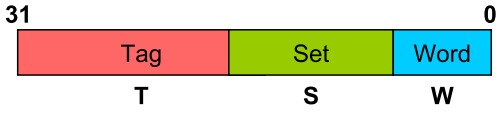

Figure 2.14: Partition of the 32-bit memory address.

The processor accesses data by specifying its address in memory. To make a correspondence between a memory address and a cache line, the former is divided into three parts: tag, set offset, and word offset (Fig. 2.14). With a cache line of 2^W bytes in size, the lowest W bits of a memory address are used as the offset into the cache line. The following S bits determine which set this address belongs to (supposing there are 2^S sets of cache lines). The remaining $32 - W - S = T$ bits (supposing the addresses are 32-bits wide) are used for the tag that are associated with each cache line to differentiate the addresses in the same set. Note that the memory address used by the cache can be physical or virtual, depending on the cache implementation [44].

As the size of a cache is very small, and if a program has a working set size larger than the cache size, it would cause a lot of cache faults. The selection of

cache lines to be removed from the cache depends on the replacement policy used. The Least Recently Used (LRU) policy, for example, replaces the least recently used lines. There are other replacement algorithms such as Least Frequently Used (LFU), First In First Out (FIFO). Also note that a cache fault can trigger the eviction of several cache lines due to the prefetching mechanism implemented in modern cache memories.

2.2.3 Associativity

The associativity of a cache indicates the number of possible cache lines where the data located at a specific memory address can be stored. It is possible to have a cache implementation in which memory data can be placed in any line. This is the case of fully associative cache. The latter is very effective in reducing collisions (cache faults). On the other hand, it is complex to implement because in order to test the existence of the data in the cache, it requires to compare the memory address against all the cache lines at the same time. Full associativity is therefore only used in small cache memories such as Translation Lookaside Buffer (TLB) caches.

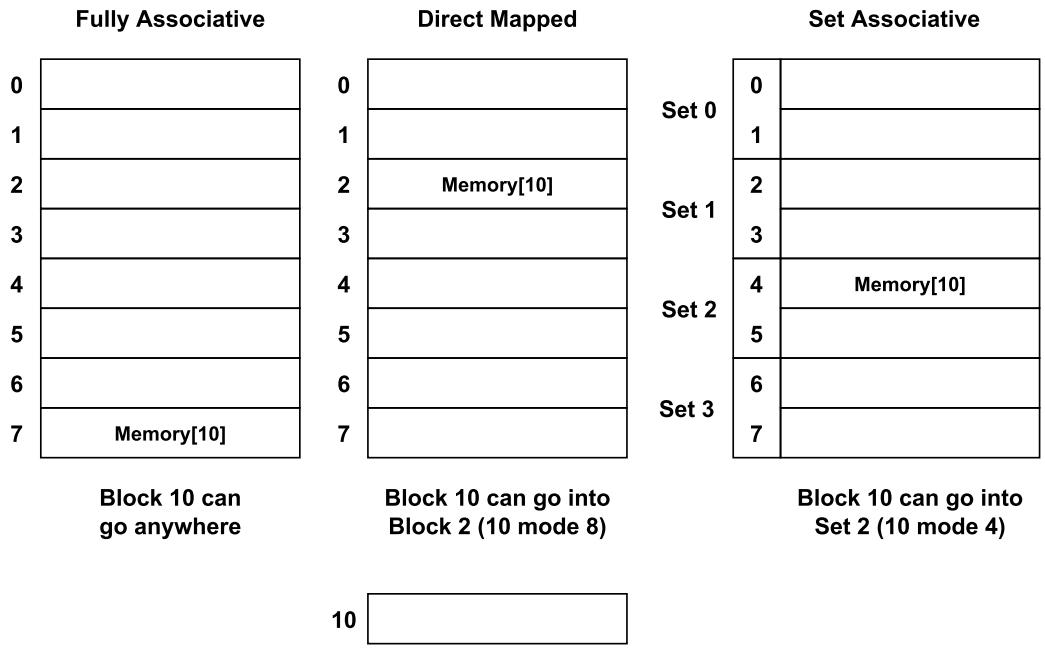

Figure 2.15: Different types of cache associativity.

For larger caches, we need a different approach. The main idea is to limit the search space. In the extreme case, a memory address corresponds to one and only one cache line. This is called the direct-mapped cache. This type of cache memory simplifies the implementation of the cache. On the other hand, the major drawback is the fact that it can create a lot of conflicting cache faults if the memory addresses used by the program are not fairly distributed.

A trade-off solution is to create a set-associative cache. A N-way associative cache divides its cache lines into N sets. The memory block corresponding to a set can be placed in any available line of that set. If there is no line left, a replacement policy is used. It's easy to see that the two types of cache we have seen above are special cases of the N-way associative cache. For the fully associative cache, there is a single set that contains all the lines in the cache and for the direct-mapped cache, the number of set is equal to the number of lines. Fig. 2.15 shows an example regarding how a memory block is placed depending on the type of cache associativity.

2.3 Non-Uniform Memory Access (NUMA)

2.3.1 The hardware view

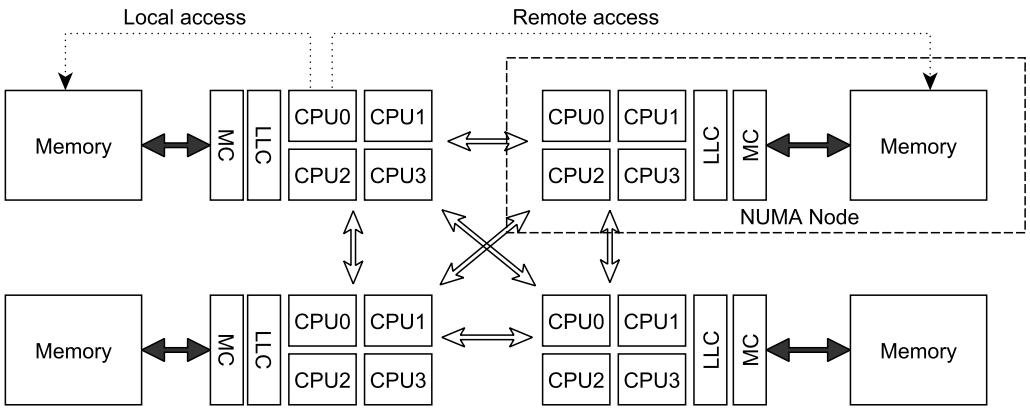

Figure 2.16: Example of a NUMA system with 4 NUMA nodes.

In a traditional multiprocessor shared memory system, all the processors are connected to the main memory via a single shared system bus. As a result, they experience the same access time to the memory. This Uniform Memory Access

(UMA) architecture has a scalability problem: as the number of processors increases, the available bandwidth for each processor decreases. To provide scalable memory bandwidth, a new architecture called Non-Uniform Memory Access (NUMA) was introduced.

In a NUMA system, there are multiple components referred as NUMA nodes. Each NUMA node may contain zero or more CPUs and memory (Fig. 2.16). The nodes of the NUMA system are connected together through some system interconnects such as AMD's HyperTransport or Intel's QuickPath Interconnect. Although a processor attached to a node can access memory of any node, the memory bandwidth and the access latency the processor experiences vary depending on the distance between its node and the node containing the memory being accessed. The lower the distance is, the higher the memory bandwidth and the lower the latency is. Each node may also have its own private caches. The cache coherency is ensured by the hardware. The described above NUMA platform is usually known as cache coherent NUMA (ccNUMA).

2.3.2 The Linux kernel view

The proposed NUMA architecture introduces interesting challenges to software developers. As the memory access time becomes non-uniform, careful attention need to be taken regarding resource placement to avoid performance loss.

Linux maintains an independent memory management subsystem for each NUMA node with memory [10]. In case of a *memoryless node*, the attached CPUs are reassigned to other nodes that have memory. Traditionally, Linux organizes memory pages into zones (e.g., normal zone, DMA zone) The zones are ordered for page allocations. It means that if a selected zone has no memory left, the page allocation is fallbacked to the zone after it. On a NUMA system, a node can have multiple memory zones and a memory zone can overlap many nodes. Linux creates for each memory zone a fallback zonelist containing different zones available across the NUMA nodes. The fallback zonelist can be ordered by zones or by nodes. Linux uses the node ordered zonelist by default: if there is a fallback, the memory allocator looks into other zones on the same node first.

The Linux scheduler is also NUMA aware. In fact, the concept of *scheduling domains* used for load balancing reflects the CPU topology in the system. The base domain spans all the cores of the physical CPU. The parent level domain

spans all the physical CPUs in the system or in case of NUMA, all the physical CPUs of a node. As Linux prioritizes local page allocations, the scheduler tries to minimize task migrations among distant nodes.

To help deal with NUMA systems more efficiently, Linux exposes the underlying NUMA topology to user space and implements several NUMA policies beside the default one. Specifically, the memory allocator can operate in 3 additional modes: *bind*, *preferred* and *interleaved* [11]. Bind mode allocates memory pages only from the set of specified nodes. Preferred mode attempts to allocate memory pages from the single specified node. If there is no memory left for allocations, it fallbacks to other nodes. Interleaved mode allocates memory pages in a round-robin way among the specified nodes. Using these policies and with the ability to control how the scheduler schedules application processes, users can find an optimal configuration for their applications.

2.4 Virtualization of Micro-architectural Components

The hypervisor enforces isolation among the VMs running on the same host system by giving a fraction of computing resources to each VM and multiplexing them on the host system. Several software and hardware techniques for virtualizing coarse-grained resources such as physical memory, processor and I/O devices were discussed in Section 2.1. On the other hand, some micro-architectural resources such as last level cache, memory controller and interconnect are difficult to be properly and/or efficiently partitioned and virtualized due to their nature in current system architectures. Regardless, they are globally shared among all VMs. In other words, they are still subject to contention, leading to performance interference. In this thesis, we consider the additional hardware features built-in modern processors (e.g., hardware performance counters) as micro-architectural components as well. There are software that are designed to exploit these hardware features. The lack of micro-architectural information being properly exposed to the VMs may cause such applications running inside the VM to work incorrectly or less efficiently. For instance, if the hypervisor doesn't properly expose the performance monitoring unit (PMU) programming interfaces to a VM, running a PMU-based profiler inside the VM doesn't produce meaningful output.

In general, depending on each type of micro-architectural resources, there are two possible strategies that can be employed by the hypervisor on standard hardware. The first one is to provide and maintain a virtualized view of the resource to the VMs using traditional virtualization techniques (e.g., emulation, paravirtualization). The second one consists of simply not exposing the micro-architectural detail and if there is contention, relying on heuristics in resource scheduling and placement to minimize the interference among the VMs. We discuss below some typical micro-architectural components and techniques proposed to virtualize them.

Performance Monitoring Unit (PMU). Modern processors have the PMU allowing software developers to do fine-grained performance profiling for their applications. The PMU consists of a set of performance counter registers that can be configured to monitor hardware-related events such as cache misses, TLB misses, clock cycles. A counter overflow interrupt is sent to the CPU when a performance counter register monitoring an event reaches a pre-defined threshold. To allow a PMU-based profiler to properly run inside a guest, the latter must have access to the performance counters and the hypervisor has to implement PMU multiplexing. The virtual PMU can be implemented for hardware-assisted guests [45, 46, 84] or paravirtualized guests [84]. In hardware-assisted virtualization, the hypervisor can allow the guest to directly access the performance counters and intercept the guest's operations as necessary via a trap-and-emulate mechanism. It can also inject virtual interrupts into the guest. The hardware can be configured to automatically save and restore the performance counters. In paravirtualization, hypercalls and a software interrupt mechanism are used to implement the virtual PMU. Moreover, some potential optimizations, such as the offsetting technique for accumulative event counters and the batching of several register configuration changes into a single call, can be used.

Dynamic Voltage and Frequency Scaling (DVFS). The DVFS component allows processor cores to operate on different frequency/voltage levels (i.e., P-states) to optimize power consumption. High-level power management (PM) policies are implemented in the OS through a system service called governor. For example, the default governor used in most systems is the ondemand governor where the frequency is adjusted according to CPU utilization. Current hypervisors can only set a single PM policy per physical core although the core may be shared among VMs with different loads. Liu et al. [76] and Hagimont et al. [60] show that this scenario makes PM inefficient and may also hurt the VM performance. Hagimont et al. [60] propose Power Aware Sched-

uler (PAS) to address the incompatibilities between virtualization and DVFS. PAS dynamically adjusts the CPU share allocated to each VM each time the processor frequency is modified. VirtualPower [82] attempts to export a set of virtualized states called VirtualPower Management (VPM) states to VMs. The PM requests from the VMs are recorded and used as inputs for VPM rules which rely on hardware scaling, soft scaling or consolidation to carry out the management decisions. Similarly, VIP [69] also define virtual P-states but per virtual CPU (vCPU) and expose them to VMs. To enforce virtual P-states, when a vCPU is scheduled out and in, its virtual P-state is saved and restored respectively.

Advanced Programmable Interrupt Controller (APIC). The APIC [26] equipped on each Intel's processor core (also known as the local APIC or LAPIC) can receive interrupts from several sources and send them to the processor for handling. In a multiple processor systems, LAPIC sends and receives the interprocessor interrupt (IPI) messages between processors. As an essential component of a processor core, the hypervisor must virtualize the LAPIC for the VMs. To virtualize the LAPIC, the hypervisor maintains a memory page called the "virtual APIC page" hosting all the virtual LAPIC's registers and then trap-and-emulate accesses to that page. Emulating guest accesses to the virtual LAPIC's control registers is not ideal in terms of performance as it requires a lot of VM Exits. To address the virtualization overheads, Intel proposed a hardware feature called APIC virtualization (APICv) that can eliminate up to 50% of VM Exits [20].

Processor caches. Current hypervisors are incapable of explicitly partitioning caches for VMs. This leads to the so-called cache contention issue where VMs compete against each other for cache usages. Several research have investigated this problem. They can be organized into two categories. The first category includes research [98, 47, 43, 65, 73, 31] which proposes to intelligently collocating processes or VMs. Concerning the second category, it includes research [89, 64, 90] which proposes to physically or softly partition the cache. The main drawbacks of these solutions are the following: cache partitioning solutions require the modification of hardware (not yet adopted in today's clouds) while VM placement solutions are not always optimal (VM placement is a NP-hard problem). Most important, VM placement solutions are not in the spirit of the cloud which relies on the pay-per-use model: why not each VM is assigned an amount of cache utilization in the same way as it is done for coarse-grained resource types?

To address this limitation, we propose Kyoto (see Chapter 3), a software technique that employs the polluter pay principle in cache partitioning. This is the first contribution in this thesis.

Non-Uniform Memory Access (NUMA). Existing approaches for virtualizing the NUMA topology fall into two categories, *Static* and *Blackbox*. The *static virtual NUMA (vNUMA)* approach is offered by major hypervisors (Xen, KVM, VMWare and Hyper-V), and consists in directly exposing to the VM the initial mapping of its vCPUs and VM memory to NUMA nodes when the VM boots. Because the OS and the system runtime libraries (SRLs, such as Java virtual machine) are unable to support resource mapping reconfiguration [42, 23], this solution can only be used if the hypervisor fully dedicates a physical CPU (pCPU) to a given vCPU (respectively a machine memory frame to the same guest memory frame), and if this mapping never changes during the lifetime of the VM. This solution is not satisfactory because it wastes energy and hardware resources by preventing workload consolidation.

On a NUMA architecture, current hypervisors are inefficient because they blindly change the NUMA topology of the guest VM in order to balance the load. The hypervisor migrates the vCPUs of a virtual machine when it balances the load on the physical pCPUs or when it starts/stops new virtual machines. The hypervisor also migrates the memory of a virtual machine when it uses ballooning or memory flipping techniques [17]. These migrations change the NUMA topology transparently to the VM [100, 103, 28]. However, guest OSs and their SRLs are optimized for a given static NUMA topology, not for a dynamic one [23]. Therefore, when the hypervisor changes the NUMA topology of the VM, the guest OS and its SRLs consider a stale NUMA topology, which results in wrong placements, and thus performance degradation.

The *Blackbox* approach was proposed by Disco [34], KVM [9], Xen, and Voron et al. [103]. It consists in hiding the NUMA topology by exposing a uniform memory architecture to the VM, and in implementing NUMA policies directly inside the hypervisor. By this way, the *Blackbox* approach can be used in case of consolidated workloads. But, as we experimentally show in Section 4.1, this approach is inefficient, especially with SRLs. A SRL often embeds its own NUMA policies, which has been proven to be much more efficient than exclusively relying on the OS (or hypervisor) level NUMA policies [56, 55, 68]. The Blackbox approach nullifies this effort because it hides the NUMA topology from the SRL. Another issue with the Blackbox approach comes from the current implementations of hypervisors, which can make conflicting placement

decisions. For instance, we have observed that the NUMA policy of the hypervisor may migrate a vCPU to an overloaded node in order to enforce locality, while the load balancer of the hypervisor migrates back that vCPU in order to balance the load (see Section 4.5).

Chapter 4 introduces extended paravirtualization (XPV), a software technique helping the whole software stack respond to changes in NUMA topology. XPV constitutes the second contribution in this thesis.

2.5 Synthesis

This chapter introduced the general context of this thesis: virtualization and several software and hardware techniques to implement it for different types of system resources. Next, we presented the cache memory architecture and the NUMA architecture that create interesting challenges at the micro-architectural level for virtualization. Finally, we summarized several micro-architectural components available in modern computer systems as well as state-of-the-art techniques proposed to virtualize each of these components. While there are many micro-architectural components that need to be addressed in virtualized environments, we believe that processor caches and NUMA are critical to the performance of VMs. Therefore, our contributions in this thesis focus mainly on processor caches and NUMA.

Chapter 3

Kyoto: Taxing Virtual Machines for Cache Usage

Contents

3.1 Motivations **30**
3.1.1 Problem statement 30
3.1.2 Problem assessment 31
3.2 The Kyoto Principle **34**
3.2.1 Basic idea: "polluters pay" 35
3.2.2 The Kyoto's scheduler within Xen 36
3.2.3 Computation of llc_cap_{act} 37
3.3 Evaluations **38**
3.3.1 The processor is a good lever 38
3.3.2 Equation 3.1 vs LLC misses (LLCM): which indicator as the *llc_cap*? 39
3.3.3 KS4Xen's effectiveness 40
3.3.4 Comparison with existing systems 42
3.3.5 Kyoto's overhead 43
3.4 Discussion **46**
3.5 Related Work **47**
3.6 Synthesis **49**

In this chapter, the micro-architectural component that we study is the last level cache (LLC). We investigate the LLC contention issue in a virtualized environment. To address this problem, a software solution called Kyoto that is inspired by the polluters pay principle is introduced. A VM is said to pollute a cache if it provokes significant cache replacements which impact the performance of other VMs. We rely on hardware counters to monitor the cache activity of each VM and to measure each VM cache pollution level. Henceforth, using the Kyoto system, the provider may compel cloud users to book pollution permits for their VMs. A VM which exceeds its permitted pollution at runtime has its CPU capacity reduced accordingly. We have implemented Kyoto in several virtualization systems including both general purpose systems (Xen and KVM) and specialized HPC systems (Pisces [86]).

3.1 Motivations

3.1.1 Problem statement

Virtualization has proved to be one of the best technology to isolate the execution of distinct applications in the same computer. The main feature which allows achieving this goal is resource partitioning. The analysis of today's hypervisors shows that only the partitioning of coarse-grained hardware resources (the main memory, the CPU, etc.) are allowed. The partitioning of microarchitectural-level components such as the Front Side Bus (FSB) and processors' caches are not taken into account, resulting in contention. This situation suits for some application types like network intensive applications. However, it is problematic for a non negligible proportion of application types. Several research have shown that contention on microarchitectural-level components is one of the main source of performance unpredictability [73]. The consequences of the latter are twofold. On the one hand, it could require supplementary tasks from cloud users. For instance, Netflix developers have reported [18] that they needed to redesign their applications to deal with this issue in Amazon EC2. On the other hand, some suggest that performance unpredictability contributes to brake the inroad of the cloud in some domains like HPC [81]. Contention on the LLC has been pointed by several research [65, 73, 31] as a critical issue.

Definition: LLC contention occurs when several VMs compete on the same LLC lines. It concerns both VMs which run in parallel (on distinct cores on the same socket) or in an alternative manner on the same core. The former situation is promoted by the increase number of cores in today's machines while the latter situation comes from time sharing scheduling. The next section presents evaluation results which attest the need to handle LLC contention.

3.1.2 Problem assessment

In order to provide a clear illustration of the issue we address, we consider the following assumptions: any VM runs a single application type and is configured with a single vCPU which is pinned to a single core.

3.1.2.1 Experimental environment

Main memory	8096 MB
L1 cache	L1_D 32 KB, L1_I 32 KB, 8-way
L2 cache	L2_U 256 KB, 8-way
LLC	10 MB, 20-way
Processor	1 Socket, 4 Cores/socket

Table 3.1: Experimental machine

All experiments have been performed on a Dell machine with Intel Xeon E5-1603 v3 2.8 GHz processor. Its characteristics are presented in Table. 3.1. The machine runs a Ubuntu Server 12.04 virtualized with xen 4.2.0.

3.1.2.2 Benchmarks

Micro benchmark. Micro benchmark applications come from [44]. In brief, a micro benchmark application creates an array of elements whose size corresponds to a specific working set size. Elements are randomly chained into a circular linked list. The program walks through the list by following the link between elements.

Macro benchmark. We use both *blockie* [79] and applications from SPEC CPU2006 [7] as complex benchmarks. They are widely used to assess the processor and the memory subsystem performance.

3.1.2.3 Metrics

The two following metrics are used: cache miss ratio (cache misses per millisecond) and instruction per cycles (IPC). The latter is used to measure an application performance. To compute these metrics, we gathered statistical data from hardware performance monitoring counters (PMC) using a modified version of perfctr-xen [84].

3.1.2.4 Evaluation scenarios

Handling an intermediate level-cache (ILC) miss takes less time (the probability to find the missed data within the other cache is high) than handling an LLC miss (which always requires main memory accesses). In the case of our experimental machine, the time taken to access each cache level (measured with lmbench [5]) is the following (approximately): 4 cycles for L1, 12 cycles for L2, 45 cycles for LLC, and 180 cycles for the main memory. Therefore, VMs can be classified into three categories: C_1 includes VMs whose working set fits within ILC (including L1 and L2), C_2 includes VMs whose working set fits within the LLC (L3), and C_3 is composed of the other applications. For each category C_i ($1 \leq i \leq 3$), we have developed both a representative and a disruptive VM, respectively noted v^i_{rep} and v^i_{dis}. Each v^i_{rep} is executed in ten situations: alone (one situation), in an alternative manner with each v^i_{dis} (three situations), in parallel with each v^i_{dis} (three situations), and both in parallel and in an alternate way with each v^i_{dis} (three situations). We report the degradation comparing to the alone execution in percentage.

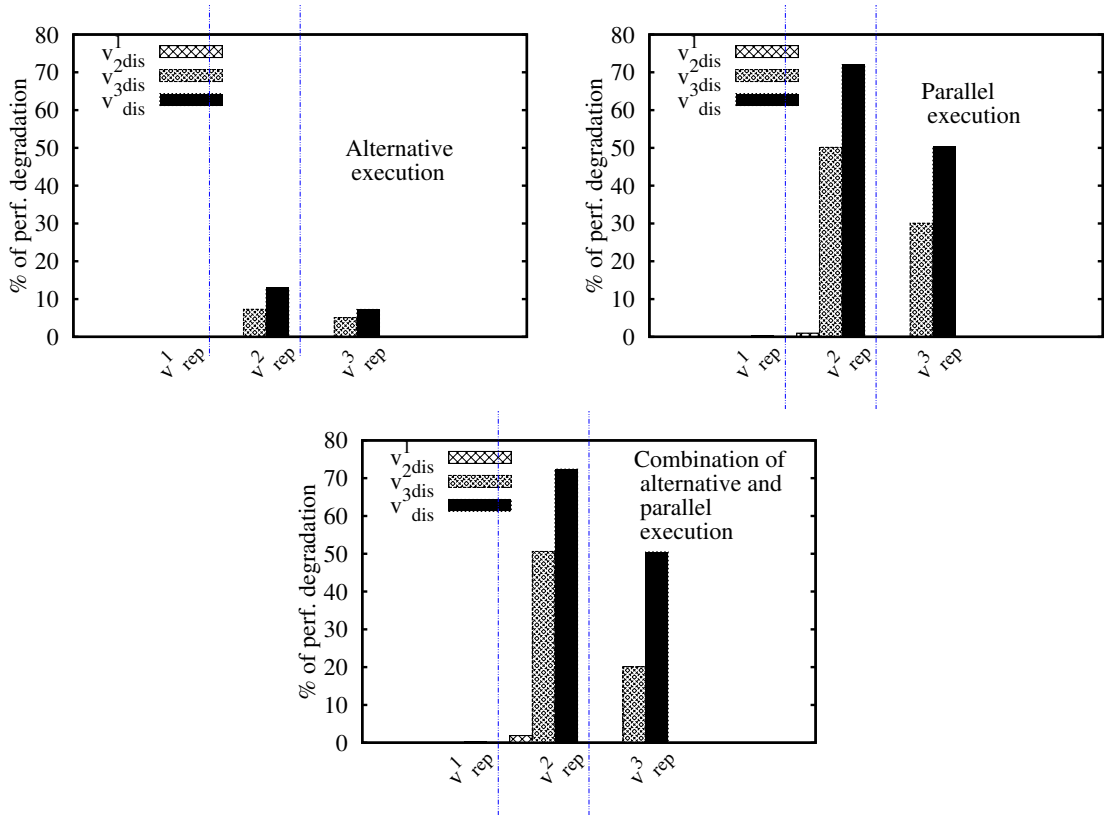

Figure 3.1: LLC contention could impact some applications.

3.1.2.5 Evaluation results

Fig. 3.1 presents the execution results of the above scenarios. Firstly, we can see that the competition on ILC is not critical for any VM type (all the first bars are invisible because the performance degradation percentage is almost nil). In addition, C_1's VMs are agnostics to both ILC and LLC contention (the three first bars of each curve are invisible because the performance degradation of v^i_{rep} is almost nil). Indeed, the cost needed to handle an ILC miss is negligible. Secondly, we can see that both C_2 and C_3's VMs are severely affected by LLC contention (the four visible bars in each curve show that the performance degradation percentage is not negligible). Thirdly, contention generated by a parallel execution is more devastating than the contention generated by an alternative execution: up to 70% of performance degradation in the former vs about 13% in the latter.

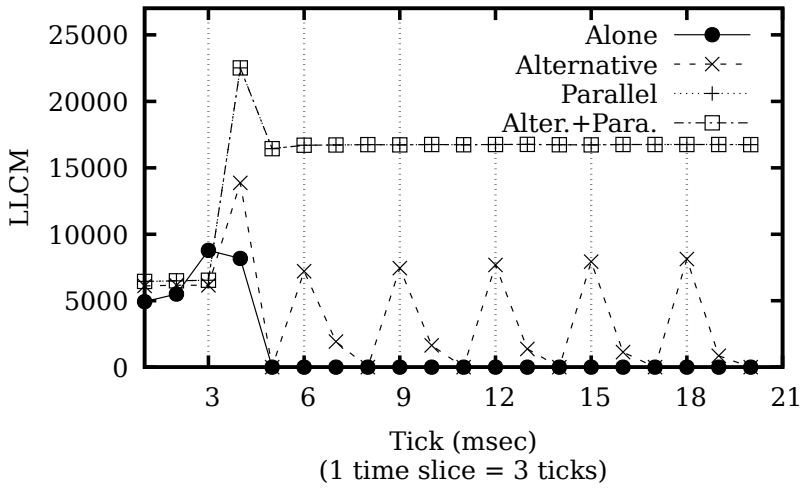

Figure 3.2: Impact of LLC contention explained with LLC misses

In order to complete the analysis, let us zoom-in on the first six v^2_{rep}'s time slices[1] (v^2_{rep} is the most penalized VM type). We can see from Fig. 3.2 that when the VM runs alone, LLC misses occur only during the first time slice (data loading). It is not the case in the other situations because of the competition on LLC lines. This problem is well observed in the alternative execution which has a zigzag shape: the first tick of each time slice is used for loading data to the LLC (because the data have been evicted by the disruptive VM during the previous time slice). Concerning the parallel execution, the cache miss rate is very high because of data eviction. This is caused by the parallel execution with the disruptive VM.

In conclusion, sharing the LLC without any partitioning strategy under its utilization could be problematic for some VMs. In this chapter we propose a solution in this direction, see the next section. In the rest of the chapter, C_2 and C_3's VMs are called sensitive VMs.

3.2 The Kyoto Principle

This section presents our solution (called Kyoto) to the LLC contention issue. After a presentation of the basic idea behind Kyoto (simple but powerful), a detailed description of its implementation within the Xen virtualization system is

[1] A time slice (30msec) is composed of 3 ticks (10msec) in Xen.

given (the patch can be downloaded at https://bitbucket.org/quocbaoit/xen-4.2.0-perfctr.git). We have also implemented Kyoto within KVM (the default Linux virtualization system) and Pisces [86] (a lightweight co-kernel for achieving performance isolation for HPC applications). An evaluation of the latter is presented in Section 3.3.

3.2.1 Basic idea: "polluters pay"

We propose a software solution whose basic idea is the same as the "polluters pay" principle of the Kyoto protocol [4]. This solution relies on the following assumptions. (1) A VM execution time results in the pollution of the LLC at a certain level. (2) Therefore, a VM which generates a high pollution level is likely to cause more contention (thus aggressive against other VMs) when it is collocated with other VMs. Under these assumptions, if one is able to instantiate a VM with a booked pollution level, and to enforce that pollution level during the overall VM lifetime, then he will have defined a solution to the problem of cache partitioning, thus cache contention. Therefore, the utilization of the LLC could be charged to cloud users in the same way as coarse-grained resources (e.g. processor, disk, main memory). This is the main idea we follow in this contribution. This idea raises two main challenges:

- How to monitor a specific VM pollution level at runtime?
- How to enforce a booked pollution level at runtime?

The first challenge can be achieved using hardware performance monitoring counters (PMCs). The latter allow to gather information about the utilization of the majority of microarchitectural-level components such as the LLC. Section 3.2.3 presents which metrics Kyoto uses to compute a VM pollution level. Concerning the second challenge, Kyoto relies on the processor, which is the central resource in a computer: a VM is only able to pollute the LLC when it is scheduled on a processor. Therefore, the processor can service as a lever to enforce a pollution level (this is illustrated in Section 3.3.1). A VM whose actual pollution level exceeds the booked one sees its computing capacity reduce. Therefore, handling the second challenge requires the extension of the hypervisor component which is responsible to schedule VM on processors. The next section presents an implementation of the Kyoto's scheduler within Xen.

3.2.2 The Kyoto's scheduler within Xen

The Kyoto's scheduler (hereafter noted KS4Xen) enforces each VM's booked pollution level during the overall lifetime of the VM. Before presenting KS4Xen, we firstly gives a quick description of the Xen credit scheduler (hereafter noted XCS), knowing that further details could be found in [39].

XCS. It is the default scheduler in Xen. It is suitable for cloud platforms since the customer books for an amount of computing capacity which should be ensured without wasting resources. XCS works as follows. A VM v is configured at start time with a credit c which should be ensured by the scheduler. To this end, the latter defines *remainCredit*, a scheduling variable, which is initialized with c. Each time a v's vCPU is scheduled on a processor, (1) the scheduler translates into a credit value (let us say *burntCredit*) the time spent by v on that processor. (2) Subsequently, the scheduler computes a new value for *remainCredit* by subtracting *burntCredit* from *remainCredit*. When the latter reaches a lower threshold, the VM is no longer allowed to get the processor. We can say that the VM is "blocked". Periodically, the scheduler increases the value of *remainCredit* for each VM blocked according to its initial credit c. This allows the VM to become schedulable.

KS4Xen. We propose KS4Xen as an extension of XCS. The former works as follows. In addition to c (introduced above), a VM is configured (booked by its owner) with the pollution level (noted *llc_cap*) it is allowed to generate during a time slice. At runtime, a scheduling variable named *pollution_quota* is assigned to each VM. As well as XCS ensures the respect of c, KS4Xen does the same for *llc_cap*. This is achieved by periodically monitoring LLC related statistics for each VM. From these collected data, the actual *llc_cap* (noted llc_cap_{act}) of each VM is computed over a period (see Section 3.2.3). The scheduler then debits the VM's *pollution_quota* according to this llc_cap_{act}. If a VM's *pollution_quota* goes negative, that VM will be in priority OVER, meaning that it cannot use the processor any more. At the end of each time slice, VMs earn a specific amount of pollution quota based on their booked *llc_cap*. If a *pollution_quota* is positive, the VM is marked UNDER, meaning that it can use the processor. There are also some codes we have introduced in order to provide a way to set a VM's *llc_cap* as a Xen command line parameter. In summary, apart from the code provided by perfctr-xen [84], which is used to collect PMCs, we made our modifications in 8 files of Xen source codes, representing about 110 LOCs.

3.2.3 Computation of llc_cap_{act}

The computation of llc_cap_{act} is periodically performed (e.g. each 100 million of instructions) for all active vCPUs. We assume that vCPUs of the same VM have the same behaviour. Therefore, only one vCPU of each VM is considered. Kyoto relies on two performance metrics: LLC Misses and UnHalted Core Cycles. Subsequently, the llc_cap_{act} is estimated using equation 3.1.

$$llc\_cap_{act} = \frac{llc\_misses \times cpu\_freq\_khz}{unhalted\_core\_cycles} \tag{3.1}$$

Being able to collect LLC related statistics is not sufficient to compute llc_cap_{act} for each specific VM. A crucial question goes unresolved: How to rightly identify PMCs of a specific VM knowing that several VMs may run in parallel atop the same LLC[2]? The Kyoto monitoring system is able to use two solutions. The first solution consists in dedicating the use of the LLC to the vCPU whose llc_cap_{act} needs to be computed. In other words, only one core in the socket is activated during the sampling time (about one billion of cycles). The other vCPUs are migrated to another socket. This solution could impact migrated vCPU performance (as shown in Section 3.3.5). The second solution comes as a response to this limitation.

The second solution relies on the use of a microarchitectural-level simulator. We have used the McSimA+ [30] simulator in our prototype. McSimA+ [30] is able to be configured to reflect a specific hardware (including processor caches, pipelines, etc.). Using a pin tool [37], the instructions generated during the execution of an application can be concurrently replayed within the simulator. McSimA+ returns PMCs related to the architecture of the machine given as the input. Relying on such a simulator, which runs atop a dedicated machine, the computation of each VM's llc_cap_{act} can be achieved following these steps:

1. KS4Xen asks the simulator to start the pin tool for a sampling period,
2. the simulator replays instructions and sends PMCs back to KS4Xen,
3. and KS4Xen computes the llc_cap_{act} based on the collected PMCs.

The next section presents the evaluation results of all KS4Xen aspects.

[2] A VM should not be punished for the pollution of another VM.

VM name	Applications
$v_{sen}{}^1$, $v_{sen}{}^2$, $v_{sen}{}^3$	respectively gcc, omnetpp, soplex
$v_{dis}{}^1$, $v_{dis}{}^2$, $v_{dis}{}^3$	respectively lbm, blockie, mcf

Table 3.2: Experimental VMs

3.3 Evaluations

After the presentation of the results which justify our choices, the evaluation of both KS4Xen's effectiveness and overhead are presented. Unless otherwise specified, any VM uses a single vCPU (having the computing capacity of a core) and runs either a SPEC CPU2006 application or *blockie* [79]. The latter is one of the most contentious application from the contention benchmark suite developed in [79]. To make reading easier, we use the following notations: v^i_{sen} and v^i_{dis} respectively identify a sensitive and a disruptive VM, ${}^{lc}v$ means that the VM v is configured with a booked *llc_cap* value equals to lc. Table 3.2 shows the name of the application which corresponds to each v^i_{sen} and v^i_{dis} ($1 \leq i \leq 3$). Throughout the rest of the chapter, the expression "we ran an application x" is equivalent to "we ran a VM hosting application x".

3.3.1 The processor is a good lever

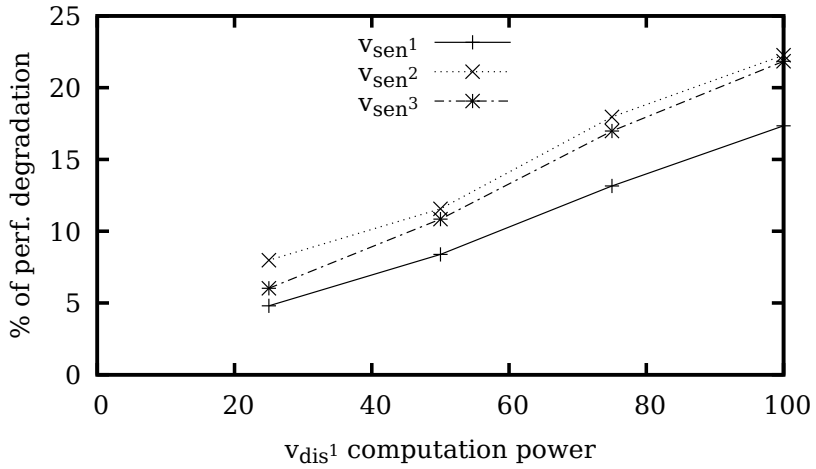

Figure 3.3: The processor is a good lever for punishing polluter/disruptive VMs

KS4Xen uses the processor as the lever to enforce an assigned *llc_cap*. The first experiment type confirms a strong relationship between a VM's computing capacity and its aggressiveness, which is correlated to its pollution level. The scenario we use for these experiments is the following. We run each v_{sen^i} in parallel with a v^i_{dis} (let us say v_{dis^1} (lbm)) while varying the computing capacity of the latter. Fig. 3.3 shows the results of these experiments. We can see that each v_{sen^i}'s performance degradation percentage linearly increases with v_{dis^1}'s capacity. Indeed, increasing v_{dis^1}'s computing capacity increases its scheduling frequency, which in turn increases its aggressiveness.

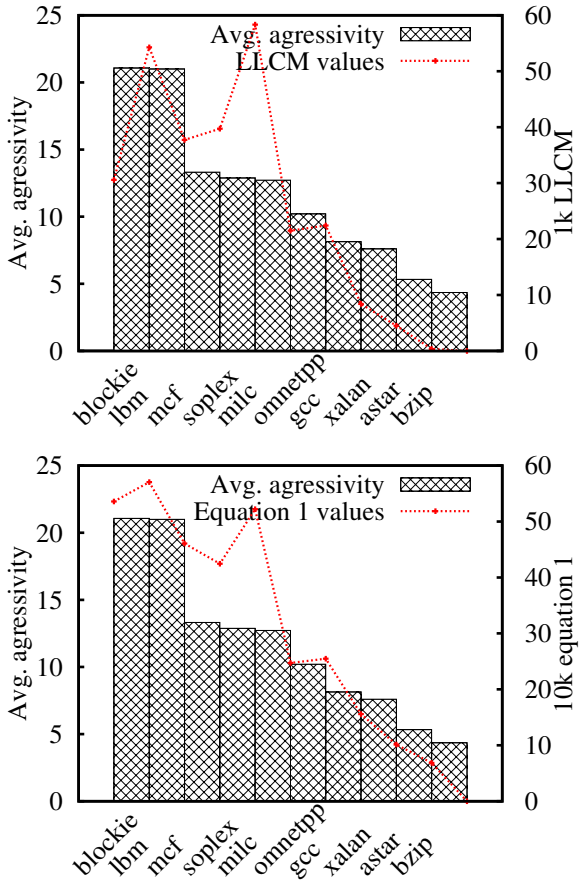

Figure 3.4: LLCM vs Equation 3.1

3.3.2 Equation 3.1 vs LLC misses (LLCM): which indicator as the *llc_cap*?

This section presents evaluation results which confirm the better accuracy of equation 3.1 (introduced in [98]) in comparison with LLCM for the estimation

of each VM pollution level. The latter can be seen as the aggressiveness level of the VM. We use the following scenario. We evaluate the aggressiveness of 10 applications (astar, blockie, bzip, gcc, lbm, mcf, milc, omnetpp, soplex, and xalan) as follows. Each application is firstly executed alone and its *llc_cap* is computed in two manners: using LLCM and using equation 3.1. Subsequently, each application is executed in parallel with each of the other applications to evaluate its real aggressiveness. The latter corresponds to the performance degradation level they causes. The average aggressiveness of each application is computed. The results of these experiments are presented in Fig. 3.4 in a descending order regarding real aggressiveness values. The latter lead to the order o_1=(blockie, lbm, mcf, soplex, milc, omnetpp, gcc, xalan, astar, bzip) while the order obtained with LLCM is o_2=(milc, lbm, soplex, mcf, blockie, gcc, omnetpp, xalan, astar, bzip) and the one obtained with equation 3.1 is o_3=(lbm, blockie, milc, mcf, soplex, gcc, omnetpp, xalan, astar, bzip). Relying on the Kendall's tau [74] method, we can see that o_3 is closer to o_1 than o_2. In conclusion, equation 3.1 is a better indicator for *llc_cap* than LLCM.

3.3.3 KS4Xen's effectiveness

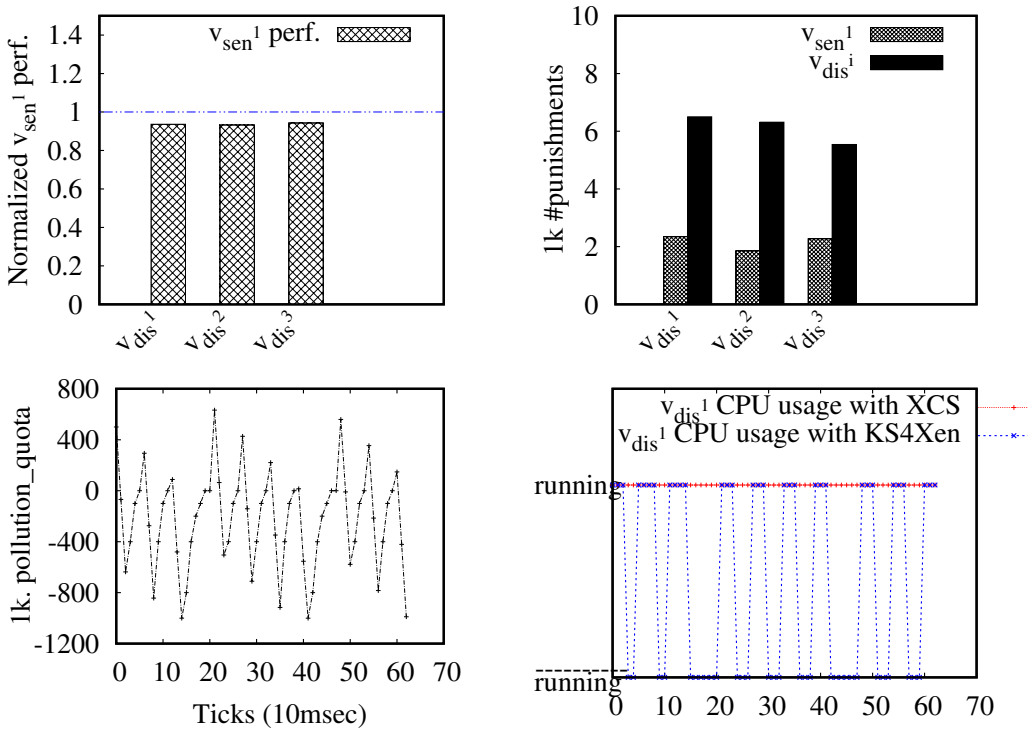

Figure 3.5: KS4Xen minimizes LLC contention, thus avoids performance variations

This section evaluates the benefits of KS4Xen in terms of LLC contention limitation. This can be judged by the ability of KS4Xen to ensure performance predictability. This evaluation is straightforward. We run in parallel ${}^{250k}v_{sen^1}$ (gcc) with different ${}^{250k}v_{dis^i}$ (lbm, blockie, and mcf). Recall that 250k is the pollution permit. Fig. 3.5 shows the results of these experiments. We can see that the performance of v_{sen^1} is almost kept whatever the aggressiveness of the concurrent VM (Fig. 3.5 top left). Fig. 3.5 top right shows respectively the number of times where v_{sen^1} and v_{dis^i} have been punished (i.e., blocked by the scheduler). All v_{dis^i} (disturber VMs) have received more penalties than v_{sen^1}. To complete the analysis, curves in Fig. 3.5 bottom plot for v_{dis^1} (lbm) respectively the variation of both *pollution_quota* and the processor utilization. Contrary to XCS (the red line), we can see that in KS4Xen, the VM is deprived of the processor for long moment every time the measured *llc_cap* exceeds the booked *llc_cap* (the zigzag line).

We have also evaluated KS4Xen scalability. To this end, we execute ${}^{250k}v_{sen^1}$ while varying the number of colocated ${}^{50k}v_{dis^i}$ (from 1 to 15 vCPUs[3]). KS4Xen is scalable if v_{sen^1}'s performance is kept. From Fig. 3.6, we can see that KS4Xen always keeps the performance of the sensitive VM whatever the number of colocated disturbers.

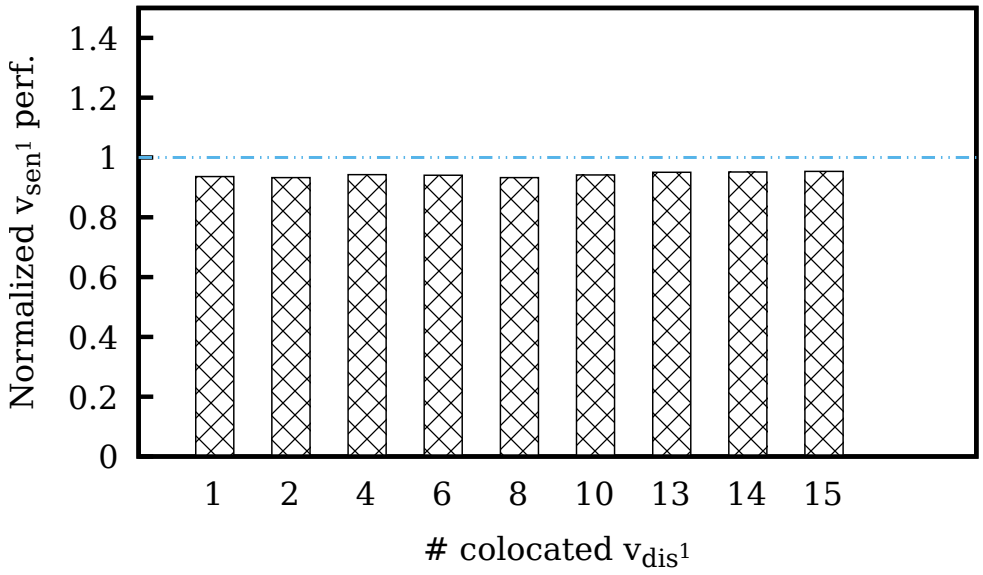

Figure 3.6: KS4Xen's scalability

[3]According to [109], the average number of vCPUs sharing the same core is about 4. Having 4 cores in our socket, we can colocate up to 16 vCPUs (remember that v_{sen^1} is already assigned one vCPU).

3.3.4 Comparison with existing systems

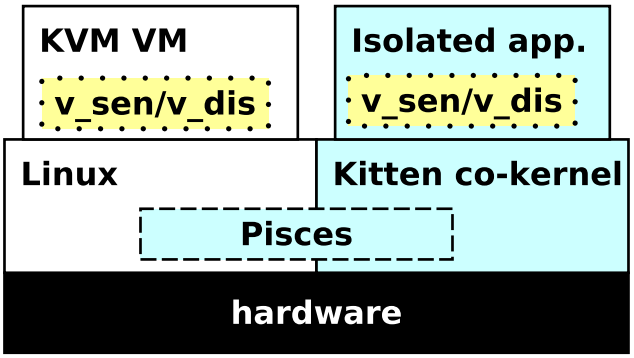

Figure 3.7: Pisces architecture

The previous section have presented the Kyoto's effectiveness in comparison with the Xen system, a general purpose virtualization system. We have also compared Kyoto with Pisces [86], a co-kernel [92] which allows building strongly isolated HPC applications (see Fig. 3.7). To guarantee performance isolation, a Pisces application runs in a VM which has the entire control of its assigned resources, without the intervention of an hypervisor. By doing so, Pisces avoids the contention within the hypervisor and other virtualization components (such as driver domains), which is known to be source of performance interference [101]. We have evaluated the Pisces capability to (1) isolate a sensitive application (v_{sen^1}) and to limit the negative effect of a disruptive application (v_{dis^1}). Subsequently, we have implemented and evaluated the effectiveness of two other Kyoto versions: one for the Linux virtualization system (via the CFS scheduler, noted KS4Linux) and the other for Pisces (noted KS4Pisces). Fig. 3.8 (the first two bars) shows that Pisces does not ensure performance predictability when the LLC is shared between a sensitive and a disruptive VM (the performance difference is about 24%). This is explained by the fact that the performance interference issue considered by Pisces is the one which comes from shared virtualization components (such as the driver domain) and coarse-grained resources (such as processors). Microarchitectural-level components like the LLC are not considered. Fig. 3.8 (the last two bars) also shows that when the previous experiment is played in a Kyoto environment, performance predictability is achieved (notice that we use the same *llc_cap* metric presented in the previous section).

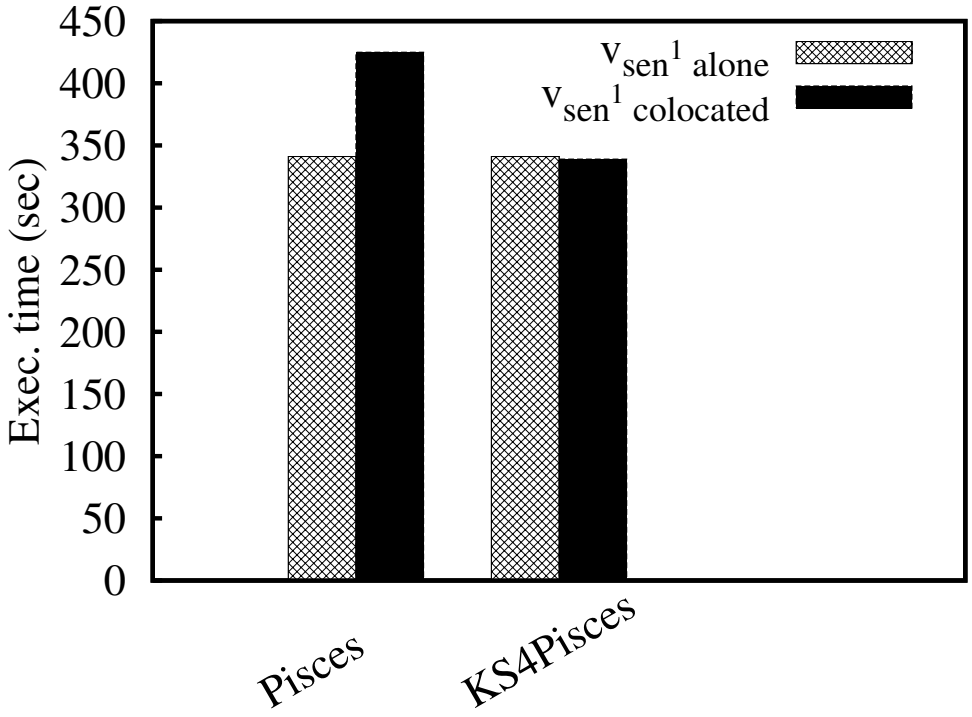

Figure 3.8: Comparison of Kyoto with Pisces

3.3.5 Kyoto's overhead

The complexity of Kyoto is $\mathcal{O}(n)$, where n is the number of vCPUs (about a hundred in data center computers) in the physical machine. This section evaluates Kyoto's overhead by relying on KS4Xen knowing the lessons learned here are applicable to other Kyoto's implementations. The execution of KS4Xen can introduce two overhead types: (1) from the solution used to identify LLC statistics related to a specific vCPU (to compute its llc_cap_{act}, see Section 3.2.3), and (2) from the monitoring system (PMCs gathering). This section evaluates the impact (if ever exists) of these overheads.

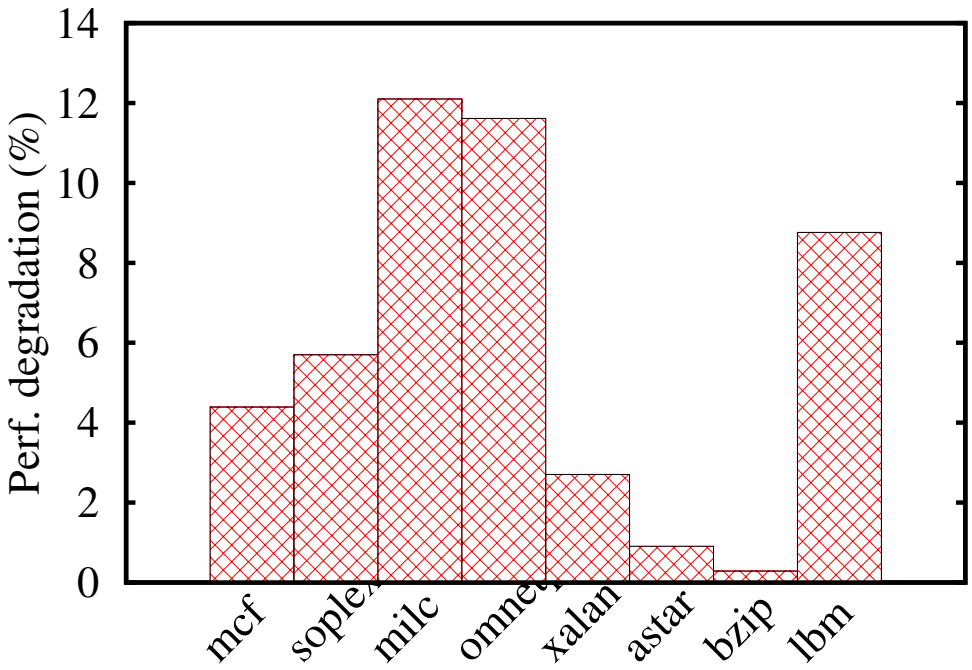

Figure 3.9: Migrating vCPU could impact VMs which host memory bound applications

llc_cap_{act} **computation**. Recall that one of the solutions used by KS4Xen to identify the LLC statistics related to a specific vCPU relies on the dedication of a socket to that vCPU for the duration of the sampling. This requires the migration of not concerned vCPUs to another socket. We evaluate the impact of this migration using the following scenario. We experiment 8 SPEC CPU2006 applications atop a NUMA machine (PowerEdge R420) composed of 2 sockets (noted $numa_0$ and $numa_1$). Each experiment uses a single VM composed of a single vCPU which starts its execution on $numa_0$. KS4Xen is configured to periodically migrate the vCPU between $numa_0$ and $numa_1$. The return migration from $numa_1$ to $numa_0$ is performed after a random period in order to mimic the time taken by KS4Xen to compute all vCPUs' llc_cap_{act}. Fig. 3.9 presents the results of these experiments. We can see that all VMs are not impacted at the same level. We have observed that the most affected applications (milc, omnetpp, lbm) are those which run memory intensive applications (up to 12% overhead). This is explained by the fact that when the vCPU is migrated to $numa_1$, all memory accesses are done remotely.

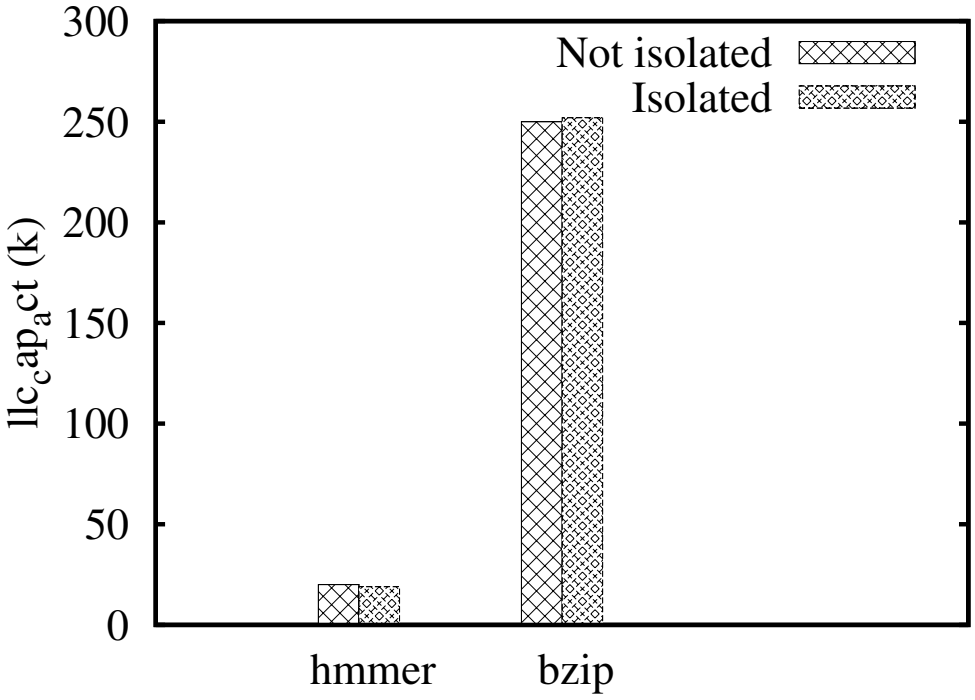

Figure 3.10: vCPU isolation could be avoided in some situations

This degradation can be minimized by reducing the number of migrations. We have identified two situations in which vCPU isolation is not mandatory. These situations are:

- A vCPU which generates a very low level of LLC misses (let us say lower than a configurable threshold) will not be isolated. Indeed, such vCPUs are neither disturbers nor sensitive. The first two bars in Fig. 3.10 shows the value of llc_cap_{act} for a VM running hmmer (known to generate low LLC misses) when its vCPU is isolated and not isolated (colocated with several disturbers vCPUs). We can see that the difference is almost nil.
- A vCPU which shares the LLC only with vCPUs which generate low level LLC misses will not be isolated. Indeed, since colocated vCPUs are not disturbers, it is most likely that the obtained llc_cap_{act} is not far from the correct value. The last two bars in Fig. 3.10 shows bzip's llc_cap is almost the same when it is colocated with several hmmer applications.

PMCs gathering. We have also evaluated KS4Xen's overhead in terms of the amount of resources it consumes. Concerning the main memory, KS4Xen

extends two data structures (*structcsched_vcpu* and *structcsched_dom*) to record PMCs for each VM. This extension is about 72 bytes, which is negligible. Concerning the processor, the execution of perfctr-xen (for gathering PMCs) is the only source of processing time consumption. To evaluate the latter, we ran in parallel two VMs which host the same CPU bound application (the SPEC CPU2006 application povray) atop the same processor. KS4Xen and XCS are experimented with different time slices (scheduling periods) to vary the intervention delay (thus the execution of the monitoring system, the potential source of overhead). Fig. 3.11 presents the results of these experiments. We can see that both KS4Xen and XCS lead VMs to the same performance level. In other words, the monitoring system used by KS4Xen does not introduce an overhead.

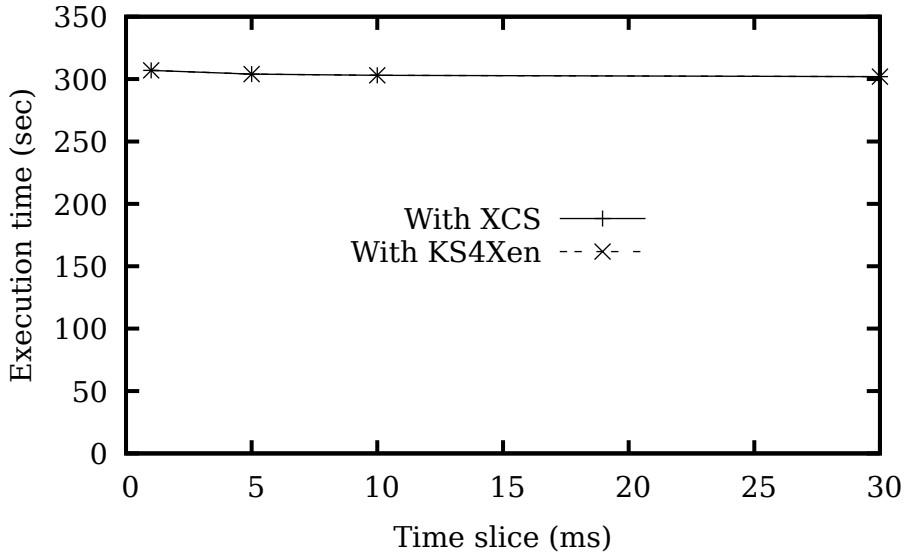

Figure 3.11: The overhead incurs by KS4Xen is near zero.

3.4 Discussion

The contribution of this chapter does not target all cloud types. It is suitable for HPC clouds since they run applications which are very sensitive to microarchitectural-level components behavior (such as LLC contention). Therefore, we assume that users of such clouds are able to deal with the new parameter we have introduced: the *llc_cap*. A question that one could ask is how the user chooses a VM's *llc_cap* value? We answer this question as follows. A cloud platform often defines a set of bookable instance types (e.g. Amazon

EC2 proposes 38 instance types[4]) which are different by the amount of resource they are assigned regarding each resource type. For instance in Amazon EC2, the particularity of a R3 instance is the fact that it is assigned a lot of memory in comparison with the computing capacity. Therefore, relying on typed VM, the provider can associate to each instance type a *llc_cap* level. We can assume that the latter is proportional to the amount of memory assigned to the instance. For instance, R3's instances will be assigned much more *llc_cap* than C3's instances since the primary needs of the latter is the computing capacity.

3.5 Related Work

Existing solutions can be organized into two categories: placement algorithms and cache partitioning.

Placement algorithms. Several prior work have proposed cache aware scheduling algorithms to address the problem of LLC contention. In the context of non-virtualized environments, [98, 47, 43, 106, 62, 59] presented some methods to evaluate the sensitivity and the aggressiveness of an application. Our Kyoto system uses one of these approaches, particularly the one presented by [98]. [99] proposed ATOM (Adaptive Thread-to-Core Mapper), a heuristic to find the optimal mapping between a set of processes and cores such that the effect of cache contention is minimized. [111] is situated in the same vein. It proposed two scheduling algorithms to distribute processes across different cores such that miss rate is fairly distributed. [49] presented a cache aware scheduling algorithm which awards more processing time to a process when it suffers from cache contention. Therefore, [49] confirms in some way the fact that the processor can serve as a lever for controlling LLC utilization as we did.

Several researches [65, 73, 31, 106, 62, 59] have pointed the problem of LLC contention in the context of virtualized environments. However, very few of them have proposed a solution to this problem. [87] studied the effects of collocating different types of VMs under various VM to processor placement schemes to discover the best placement. The main limitation of this solution is the fact that it needs to know the applications which are running within VMs (to evaluate the collocation effects). [29] proposed a cache aware VM consolidation algorithm which chooses the consolidation plan so that the overall

[4] https://aws.amazon.com/ec2/instance-types/

LLC misses are minimized in the IaaS. This solution considers the entire IaaS, not a single machine as we did.

Cache partitioning. In this category we can distinguish two main approaches. The first approach is based on cache replacement policies. It is independent from the execution environment (virtualized or not). According to this approach, [89, 64] proposed a dynamic insertion policy (DIP) which adapts the insertion policy (LRU or BIP) according to process memory activities. By doing so, DIP avoids to keep in the cache data of a VM which is parsing a large working set (a kind of disruptive VM). This solution is limited to a single category of disruptive VMs. [47] trends in the same direction by proposing PD (Protecting Distance), a cache replacement policy which protects cache lines that may be reused. [108] proposes a cache management policy called PIPP (Promotion/Insertion Pseudo-Partitioning). The latter partitions the cache by managing both cache insertion and promotion policies. [90] presents UCP (Utility-based Cache Partitioning), a runtime mechanism for partitioning the cache between multiple applications. UCP monitors each application using a cost estimation hardware circuit. Collected data are used by a partitioning algorithm to decide the amount of cache resources to allocate to each application. The policy is implemented through hardware and software modifications. [63] presented a QoS enabled cache architecture which enables more cache resources for high priority applications. Applications are assigned a priority level (this is comparable to our *llc_cap*). Then each cache line is tagged with a priority level.

The second approach addresses the cache contention issue using software based cache partitioning. Our solution uses this approach. [67, 104] proposed to partition the cache using page coloring [110]. Each VM is reserved a portion of the cache, and the physical memory is allocated such that a VM cache lines map only that reserved portion. This idea is very nice but difficult to implement. It depends on both the architecture of the cache and the replacement policy. Moreover, allocating physical pages to enforce the use of a specific place of the cache could be difficult to implement without wasting memory resources. For these reasons, [67, 104] only presented preliminary results.

Positioning of our work. The main drawbacks of the above solutions are the following: cache partitioning solutions require the modification of hardware while VM placement solutions are not always optimal (VM placement is a NP-hard problem), **most important these solutions are not in the spirit of the cloud** which relies on the pay-per-use model: why can't we allow each VM

to book for an amount of cache utilization such that the virtualization system ensures it in the same way as it does for other coarse-grained resource types (CPU, memory, etc.). In this work, we have proposed the Kyoto system which is a step in that direction.

3.6 Synthesis

We presented in this chapter a new approach to address the issue of performance unpredictability due to LLC contention in a virtualized cloud environment. Our approach is inspired by the polluters pay principle which is applied as follows: any VM should pay for the amount of pollution it generates in the LLC. To implement it, we relied on hardware counters to monitor the utilization of the LLC by VMs, and we implemented a new vCPU scheduler which enforces at runtime a booked pollution level of a VM. We have presented a prototype for Xen system, KVM and Pisces. These prototypes have been evaluated using reference benchmarks (SPEC CPU2006), showing that they can enforce performance isolation between VMs even in case of LLC contention.

Chapter 4

When eXtended Para–Virtualization (XPV) Meets NUMA

Contents

4.1 Motivations **51**
4.1.1 Description 52
4.1.2 Limitations 52
4.1.3 Synthesis 56
4.1.4 Hot-(un)plug as a solution? 56
4.2 eXtended Para-Virtualization **57**
4.2.1 Principle 57
4.2.2 Methodology for making legacy systems XPV aware 58
4.3 Technical Integration **61**
4.3.1 Xen modifications 62
4.3.2 Linux modifications 63
4.3.3 Application-level modifications: the HotSpot Java virtual machine use case 64
4.4 Evaluations **65**
4.4.1 Experimental setup 65
4.4.2 XPV implementation efficiency 66
4.4.3 vNUMA vs blackbox solutions 68
4.4.4 XPV facing topology changes 70

4.4.5 Automatic NUMA Balancing (ANB) limitations . . 73
4.4.6 XPV internals 74
4.5 Related Work **75**
4.5.1 Industrial solutions 75
4.5.2 Academic solutions 76
4.5.3 Positioning of our work 76
4.6 Synthesis **77**

In this chapter, we are interested in studying the NUMA micro-architectural component. We propose a new principle called extended paravirtualization (XPV) to efficiently virtualize a NUMA architecture. XPV extends the well known paravirtualization (PV) principle in two directions. First, in the same way as PV actually abstracts away I/O devices [40], XPV extends the principle by also abstracting away the physical NUMA topology into a virtual NUMA topology that can change at runtime. Second, while currently PV is only used at the guest OS level to implement optimized drivers for virtualized environments, XPV extends this principle to the software runtime libraries (SRLs, such as Java virtual machine). By doing so, XPV allows each layer in the virtualization stack to implement what it does best: optimization of resource utilization for the hypervisor and NUMA resource placement for the guest OS and the SRLs.

4.1 Motivations

Several research work realized in native environments [75, 53, 52, 56] have demonstrated the necessity to specialize system software (e.g., OS and SRL) with respect to the NUMA architecture. The same observations were made [103, 78, 23] in virtualized environments. Among the existing solutions (presented in Section 4.5), vNUMA is the most promising one. This section presents vNUMA and the motivations for our work. Although the problem we address affects all hypervisors (as far from our knowledge), we use Xen for the assessment.

4.1.1 Description

vNUMA consists in showing to the VM a virtual NUMA topology which corresponds to the mapping[1] of its virtual resources on physical NUMA nodes. By this way, there is no need for the hypervisor to include a NUMA optimization algorithm since the optimization can be performed by the guest OS. At the time of writing of the thesis, the most popular hypervisors (Xen, VMware, hyper-V and VMware) implement vNUMA. This is considered as the ideal approach to handle NUMA in virtualized environments because it makes guest OS's NUMA policies (assumed to be the most optimal ones) effective, as argued by VMware in [23][2].

4.1.2 Limitations

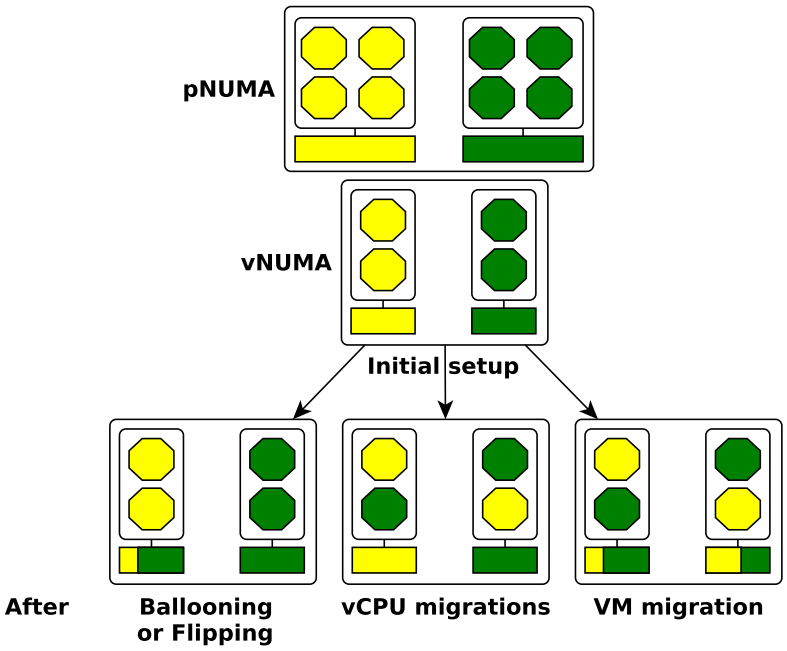

Figure 4.1: Hypervisor's resource utilization optimizations may lead to vNUMA changes.

[1] Notice that the VM does not see the full machine NUMA topology.

[2] "*... Since the guest is not aware of the underlying NUMA, the placement of a process and its memory allocation is not NUMA aware... vSphere 5.x solves this problem by exposing virtual NUMA topology for wide virtual machines.*" by VMware in [23]

The common implementation approach of vNUMA consists of the hypervisor storing the virtual topology of the VM in its ACPI tables, so that the guest OS uses it at boot time as any OS does. This implementation has the advantage to be straightforward. However, its main limitation is that a change in the NUMA topology cannot be taken into account without rebooting the VM [23]. In fact, existing OSs are not designed to dynamically take into account NUMA topology changes. Also, a simple solution based on hot-(un)plug of CPU and memory resources is not suitable as its might seem, more details in Section 4.1.4. Topology changes occur in a virtualized system because of resource utilization fairness and optimizations (to avoid waste) implemented by the hypervisor. To achieve these goals, the hypervisor is allowed to dynamically adapt the mapping between the vCPUs and the pCPUs, and the mapping between the guest physical addresses (GPAs) and the NUMA nodes, by changing the mapping between the GPAs and the host physical addresses (HPAs). More precisely, the hypervisor can change the NUMA topology of a VM due to the following decisions (summarized in Fig. 4.1).

CPU load balancing. Resource overcommitment is the widely used approach for optimizing resource utilization in virtualized datacenters. It consists in allowing more resource reservation than the available resources. For the CPU, this approach could lead to load imbalance, thus unfairness. This issue is addressed by the hypervisor by migrating vCPUs from heavily contended nodes to less contended ones. Such migrations could be frequent due to temporary CPU load imbalance as indicated by VMware [23]. For instance, we have observed that the execution of three 15vCPUs/12GB SpecJBB 2005 VMs on a 8-node machine - 6 cores per node (two nodes dedicated to the privileged VM in Xen) - generates about 20 vCPUs migrations between different NUMA nodes per minute. We voluntarily choose large VMs to observe the generated migrations.

Memory ballooning. Ballooning is the commonly used technique to implement memory overcommitment. It allows dynamic memory reclaim (when the VM does not use its entire memory) and allocation (when the VM needs memory) from/to a VM. Ballooning may induce a modification of the mapping between GPAs and HPAs. Pages which are reclaimed (on balloon inflate) can be given (on balloon deflate) on any NUMA node.

Memory flipping. Memory flipping is the recent approach used by most hypervisors to implement zero-copy during I/O (e.g., network) operations [38, 6]. It consists in exchanging (instead of copying) memory pages between the

driver VM (the one which includes device drivers) and the user VM which sends/receives packets. When the driver VM and user VMs are located on different NUMA nodes (which is commonly the case), user VMs which perform I/O operations will see a portion of their memory remapped on the driver VM's node. To assess this issue, we ran BigBench [54] on our 8-node machine. All the VM resources are initially mapped on a single NUMA node which is distinct from the one used by the driver VM. Without rebooting the VM, we repeated the execution of BigBench 8 times. Fig. 4.2 shows the amount of remote memory (due to memory flipping) at the beginning of each execution of the benchmark. We can see that it increases in respect with the number of execution, up to 45% of the VM's memory is remote at execution number 5 due to the memory flipping mechanism. This degrades the application performance by 32%.

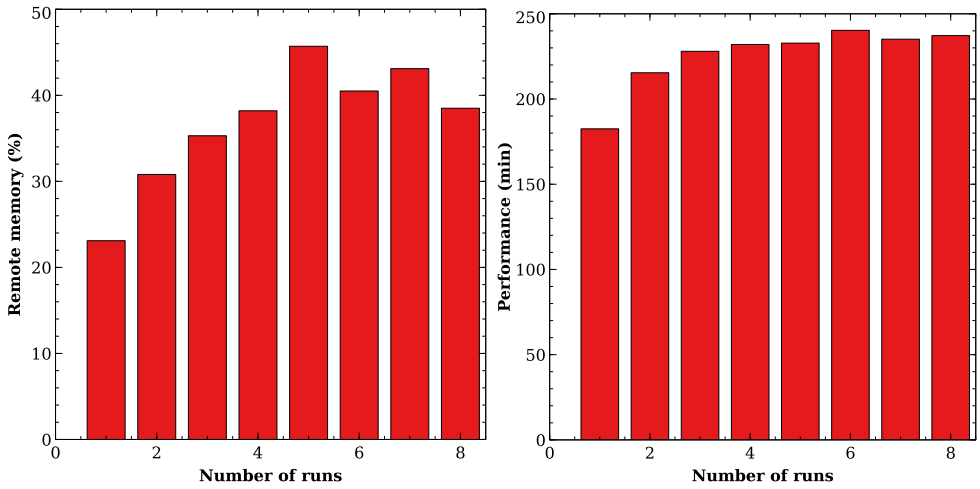

Figure 4.2: The effect of memory flipping, which occurs on I/O intensive applications. The top curve shows the proportion of the VM's memory which has been remapped on remote nodes after each run. The bottom curve shows the performance of the application after each run.

VM live migration. VM live migration involves moving a running VM from one physical host to another one. It is a central technology in today's datacenters. For instance, dynamic VM packing (which is a common approach used to optimize resource utilization), physical server failure handling, datacenter maintenance, overheating power supply management and hardware upgrades rely on it [95]. All of this makes VM live migrations very frequent in the datacenter, especially large ones (such as Google Cloud Engine [95]). The migration of a VM may change the mapping of its vCPUs and GPAs on the destination machine.

We assess in Xen the impact of using a stale topology as follows. We used two memory intensive benchmarks: STREAM [80] (a synthetic benchmark measuring the sustainable memory bandwidth) and LU from Spec MPI 2007 [14]. The benchmarks run within a VM (called the tested VM) configured with 12 vCPUs and 30GB memory. They use TCMalloc [68] as the SRL. We compared three situations: (1) the VM sees a UMA topology but the hypervisor implements the interleaved policy (the VM's memory is interleaved by a granularity of 1GB among NUMA nodes selected to host the VM), which corresponds to the default Xen solution. (2) the VM sees a NUMA topology which corresponds to its exact resource mapping (noted vNUMA); (3) and the VM sees a NUMA topology which is different from its actual mapping: all of its CPUs are migrated away from the initial nodes (noted Stale vNUMA). The initial resource mapping of the VM is as shown in Fig. 4.1 while the stale topology is the one caused by the transition labeled "vCPU migration". The experiment results are shown in Fig. 4.3. Lower is better for LU (the left curve) while it is the opposite for STREAM (the right curve). When we compare Interleaved and vNUMA (first two bars), we can observe that static vNUMA is the most efficient configuration. For instance, we can notice for LU up to 13% of performance difference. However, Interleaved becomes better than vNUMA when the initial topology becomes stale.

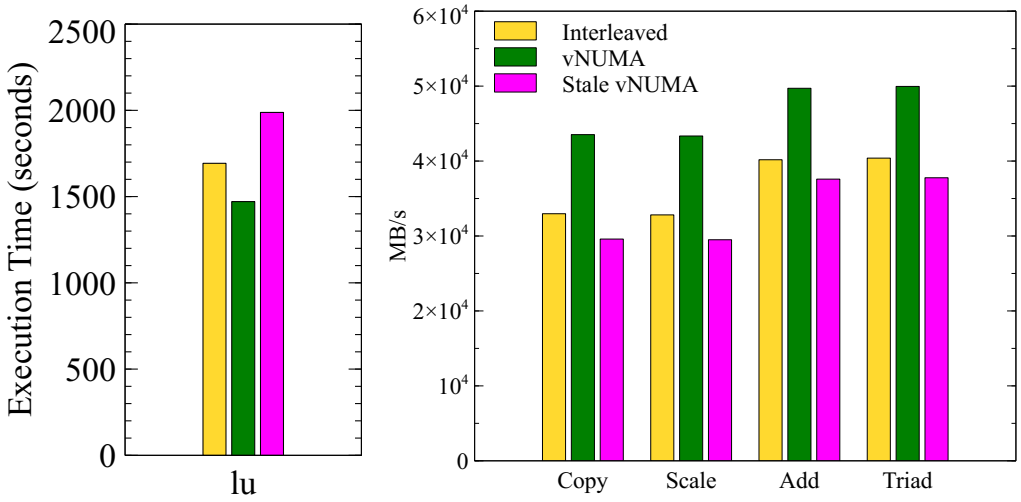

Figure 4.3: Assessment of the negative impact of presenting a stale topology to the VM when vNUMA is used. Evaluation realized using LU from Open MPI 2007 (left, lower is better) and STREAM (right, higher is better).

4.1.3 Synthesis

Although vNUMA is the ideal approach for taking into account NUMA in virtualized environments, its current implementation makes it static, thus inefficient facing topology changes. Hypervisor vendors like VMware and Hyper-V have also underlined this limitation. Typically, the VMware documentation [23] states: *"The idea of exposing virtual NUMA topology is simple and can improve performance significantly. [...] On a virtual environment, it becomes likely that the underlying NUMA topology of a virtual machine changes while it is running. [...] Unless the application is properly reconfigured for the new NUMA topology, the application will fail. To avoid such failure, ESXi maintains the original virtual NUMA topology even if the virtual machine runs on a system with different NUMA topology."*. Today, to prevent the inefficiency of vNUMA, hypervisor providers make the following recommendations [51, 71, 41, 66, 2, 102, 96]: (1) either the data center operator enables vNUMA and disables all hypervisor's resource management optimizations, which leads to resource waste or unfairness, (2) either she disables vNUMA and keeps hypervisor's resource management optimizations by using blackbox solutions (the hypervisor implements NUMA policies and exposes a UMA architecture to the VM), which are not optimal for performance (see Section 4.4). None of these two solutions is satisfactory.

4.1.4 Hot-(un)plug as a solution?

One might imagine the utilization of resource hot-(un)plug as a solution for providing an adaptable vNUMA solution. Although this solution is without any doubt elegant, it is not as straightforward and complete. These are the main reasons:

1. Resource hot-(un)plugging is only possible for CPUs or memory which have been discovered and recorded by the guest OS at boot time [42]. Therefore, the implementation of a hot-(un)plug based solution needs to first show to the VM at boot time the entire physical machine topology, knowing that the actual VM's resource mapping concerns only a subset of the topology. This requires deep kernel code rewriting in both the VM's OS setup code and also the hypervisor code which is responsible for starting a VM. The development cost[3] of this step is very high compared

[3]Notice that we tried unsuccessfully this alternative during several months.

to the approach we present in the next section. Moreover, it would be difficult to port this code to other systems.

2. Assuming that the above is implemented, memory hot-(un)plug is only possible at the granularity of a block (e.g., 512MB in the current Linux kernel version). Remember that topology changes could be caused by the relocation of very few memory pages (e.g., see memory flipping for example).

3. Finally, Hot-(un)plug is not sufficient because the SRL which runs inside the VM is not aware of the new topology.

For all these reasons, hot-(un)plug is considered incompatible with vNUMA, which is also the VMware opinion [42, 23]. In this chapter, we present eXtended Para-Virtualization, a principle for implementing an adaptable vNUMA.

4.2 eXtended Para-Virtualization

This section describes the eXtended Para-Virtualization (noted XPV) principle and our methodology to implement it in legacy systems.

4.2.1 Principle

In this contribution, we propose to make the static vNUMA approach dynamic by revisiting the interface between the hypervisor and the VM. At high level, the hypervisor exposes a dynamic virtual NUMA topology, which abstracts away the physical NUMA architecture. Because current OSs and SRLs are unable to handle a dynamic NUMA topology, we propose the *eXtended Para-Virtualization* (XPV) principle to dynamically adapt the NUMA policies used in the kernel of the guest OS and in the SRLs when the NUMA topology of the VM changes.

XPV extends the para-virtualization (PV) principle to the whole hardware and the SRLs. PV consists in modifying the code of the kernel of the guest OS to efficiently virtualize I/O devices (the split-driver model), virtualize the MMU, virtualize the time, enforce protection and handle CPU exceptions. We extend the PV principle to the whole hardware by also virtualizing the

NUMA topology. Instead of using a driver that considers a static NUMA topology[4], the guest kernel uses a para-virtualized driver that considers a dynamic NUMA topology. We also extend the PV to the SRLs. We propose to modify SRLs with a para-virtualized NUMA management driver tailored for virtualized environments. Instead of considering a static NUMA topology, this para-virtualized NUMA management layer adapts the NUMA policy of the SRL when the NUMA topology of the VM changes. Notice that the hypervisor works as usual for non-XPV-aware VMs, meaning that XPV- and non-XPV aware VMs can share the same host. This is also true for XPV- and non-XPV aware applications which run inside a XPV aware VM.

By definition, XPV requires modifications in both the guest kernel code and the SRL code. In the remainder of the section, we present a systematic methodology for applying XPV to different legacy systems, and we show that the required modifications remain modest.

4.2.2 Methodology for making legacy systems XPV aware

In the hypervisor, we implement XPV by adding a notification to the guest kernel when the NUMA topology changes. In the guest kernel, we implement XPV mainly by adapting the NUMA-aware components and by forwarding the notification to the SRL when the guest kernel receives a notification from the hypervisor. In the SRL, we implement XPV mainly by adapting the NUMA-aware components when the SRL receives a notification from the guest kernel. Fig. 4.4 presents the components modified to implement XPV, and the remainder of the section details these modifications.

[4]The code that handles NUMA in current OSs is often spread in the kernel. In order to simplify the presentation, we consider that this code forms a driver.

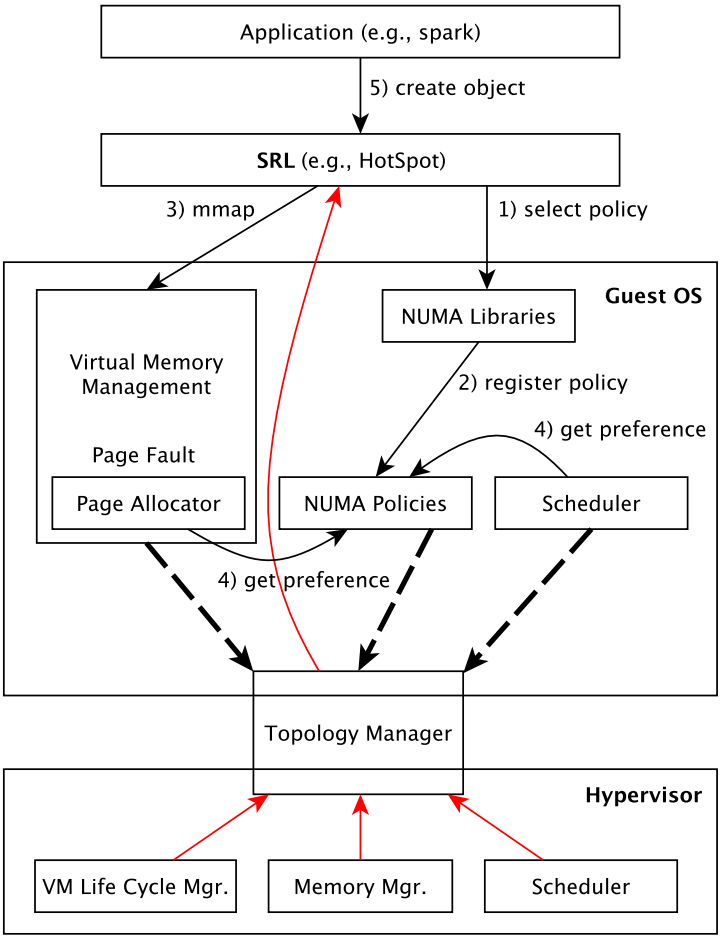

Figure 4.4: Overview of the components involved in the XPV implementation. Solid black arrows present the different steps (1-5) required to handle a dynamic virtual NUMA topology in SRLs. Dashed arrows show guest kernel components involved when the NUMA topology of the VM changes. Solid red arrows represent the notifications when the NUMA topology changes.

Design choices. When the hypervisor creates a VM, it gives the VM a topology through ACPI tables. The static vNUMA approach exposes a NUMA topology while the blackbox solutions expose a UMA one. Exposing an initial NUMA topology to the VM is inadequate for XPV because:

- Many NUMA-aware kernel components in the guest are hard to modify to take into account changes in the NUMA topology.
- Such kernel components may introduce unpredictable performance if they implement NUMA optimizations on a wrong topology.

Instead, XPV makes the following design choices:

- We expose a UMA topology through ACPI tables so that kernel components will not implement NUMA optimizations.
- We expose the real NUMA topology through a different data structure shared between the hypervisor and the VM. This data structure is updated upon resource remapping by the hypervisor. Key kernel components in the guest which are critical are modified to use this (dynamic) NUMA topology. We modified three components: the page allocator, the scheduler and the NUMA policy manager.

Hypervisor layer adaptation. We introduce a new driver, called the *topology manager*. This driver follows the split-driver principle: a part of the driver is implemented in the hypervisor while the other part is implemented in the guest kernel. The topology manager maintains the NUMA topology of each VM and, when the NUMA topology changes, notifies the VM with an interrupt. As exposing a UMA topology is the default behavior of any hypervisor, implementing the topology manager is straightforward. It requires modifications in the VM life cycle manager of the hypervisor in order to create and destroy the data structure associated to a VM, in the memory manager in order to record the physical addresses used by a VM and in the scheduler in order to record the location of the vCPUs. These components are easy to identify in any hypervisor and we were able to implement XPV in Xen and Linux KVM.

Guest kernel adaptation. The implementing XPV requires first the implementation of the kernel part of topology manager driver. The kernel uses this driver at bootstrap to retrieve the initial topology of the VM and to initialize its NUMA-aware data structures. Then, when the driver receives a notification from the hypervisor, it increments a counter used by the SRLs to know the current version of the NUMA topology (see below), retrieves, through a shared memory, its actual NUMA topology (e.g., physical location of the vCPUs) and updates the NUMA-aware data structures of three components of the guest OS: the page allocator, the NUMA policy manager and the scheduler. As a consequence of this update, the kernel considers the new NUMA topology for the newly allocated pages and the newly created threads. However, the kernel does not try to relocate the previously allocated pages and threads: instead, we let kernel's NUMA policies (such as Automatic NUMA Balancing in Linux) and the SRL migrate its memory and its thread because we consider that they are those who know how to efficiently handle the new NUMA topology.

To summarize, implementing XPV in the guest kernel requires the implementation of the topology manager and modifications in bootstrap code, in the page allocator, NUMA policies and the scheduler. These components are easy to identify in any kernel and we were able to implement XPV in Linux and FreeBSD.

SRL layer adaptation. For the adaptation of the SRL layer, we need to consider the system libraries/APIs (i.e., the `c` and `numa` libraries) and SRL itself (e.g., jemalloc [3] or HotSpot [1]). For the former, only two modifications are required: (1) the modification of the functions that retrieves the NUMA topology in order to use our topology manager driver instead of the static ACPI tables, and (2) the addition of a new function retrieving the topology version number. The latter is useful for SRL to identify that its internal NUMA topology becomes stale.

Each SRL has to implement its own algorithm to handle a NUMA topology change. However, we have identified that, in our three studied SRLs (HotSpot [1], TCMalloc [68], and jemalloc [3]), we can apply a systematic methodology to implement XPV. This methodology consists in modifying the code of three components that are often found in SRLs: the bootstrap code, the thread manager and the memory manager. In the bootstrap code, the SRL has to record the initial topology version number of the topology. In the thread manager, the SRL may migrate the threads when the topology changes (i.e., when the topology version number changes) to ensure locality with memory. In the memory manager, the SRL has to update its internal data structures when the topology changes, and, if required, to adequately migrate the pages that are already allocated to the application.

4.3 Technical Integration

We applied the methodology described in the previous section in two legacy hypervisors (Xen and KVM), two legacy guest OSs (Linux and FreeBSD) and three legacy SRLs (HotSpot, TCMalloc, and jemalloc). Table 4.1 summarizes the efforts required for each legacy system. All the patches are available at `https://github.com/bvqbao/numaVirtualization`.

Systems	#files	#LOC
Xen 4.9	8	117
KVM from Linux 4.14	6	218
Linux 4.14	26	670
FreeBSD 11.0	23	708
HotSpot 8	3	53
TCMalloc 2.6.90 (gperftools-2.6.90)	3	65
jemalloc 5.0.1	9	86

Table 4.1: XPV integration in several legacy systems.

4.3.1 Xen modifications

When Xen creates a VM, given the requested and available resources, it first determines on which NUMA node(s) to place the VM. If more than one NUMA node are needed, Xen statically allocates memory to the VM in a round-robin way (with 1GB granularity by default) over the selected NUMA nodes. The latter are usually referred to as the VM's home nodes. Xen also sets the soft affinity of the VM's vCPUs to pCPUs of the home nodes. This soft affinity is a preference and does not prevent the migration of vCPUs to different nodes when the home nodes are overloaded.

In order to implement XPV in Xen, the topology manager first records the initial topology of the VM when the latter boots. This initial topology is stored in a memory region shared between Xen and the guest kernel. Recall that topology changes may happen on the following cases: vCPU migrations (due to vCPU load balancing) inside a machine, grants acquisition on new memory pages (due to memory flipping), memory page migrations inside a machine (due to ballooning), and VM migrations between machines. For the three former cases, the modification performed in Xen is straightforward. Whenever a vCPU of a VM is migrated to a different NUMA node, we update the corresponding VM's topology in the shared memory region and notifies the VM by injecting an interrupt in the guest. Regarding page migrations, in Xen, they take the form of page deallocation/reallocation as a consequence of memory ballooning. The new location(s) of memory pages are taken into account as follows: when the migrated pages return to the guest OS, the latter then examines on which node each page is located and puts the pages into the correct free page lists (see Section 4.3.2, Memory allocator). The same operation is realized when the VM acquires grants on pages located on new nodes. In fact, no modifications at Xen level are needed in these cases. Concerning changes caused by the migration

of the VM, they can be considered as a combination of vCPU migrations and memory migrations. Therefore, we combine the above techniques together for handling such changes.

4.3.2 Linux modifications

Page allocator. The major challenge of implementing XPV in an OS is to adapt the page allocator. In Linux, memory pages on each node are divided into zones. Linux relies on the NUMA policy used by the application to search for free pages. The page allocator applies the NUMA policy to select a node and then a zone in that node for page allocations. Each zone has a component called the buddy system [72] which is responsible for page allocations inside the zone. The buddy system breaks memory into blocks of pages and maintains a separate page list for each block size. Most of the time, allocation requests are for single page frames. In order to get better performance, a per-CPU page frame cache is used to quickly serve these requests. Allocations of multiple contiguous pages are directly handled by the buddy system.

We start with a UMA version of the page allocator and we modify it to be NUMA-aware as follows. Firstly, we add new variables representing the actual NUMA information used by application processes. Secondly, we partition the per-CPU page frame cache and the free page lists of the buddy system by the number of physical NUMA nodes. The kernel can know which page comes from which node because it knows the machine frame number of each page and the memory range on each physical node. In contrast with normal NUMA systems, with XPV, the page allocator will apply the NUMA policy (stored in the newly added variables) to determine the allocated node *after* a zone is selected. To improve performance, the mapping from a page number to a physical node can be cached and will be updated only at points where page migrations may occur (e.g., ballooned pages returning to the memory allocator).

Scheduler. Linux scheduler organizes CPUs into a set of scheduling domains for load balancing. A scheduling domain consists of a set of CPUs having the same hardware properties regarding their location in the NUMA topology. As a result, they form a tree-like structure. We could have reproduced such structure in the scheduler with the real NUMA topology. However, in XPV, for simplicity, we only builds a single domain containing all the CPUs of the VM. We found that this is actually an acceptable solution as we could avoid the adjustments in the scheduler when the topology get changed.

NUMA policy manager. We only describe here the Automatic Numa Balancing (ANB) [58] as it is the most advanced and complex NUMA optimization is Linux. Once activated, ANB periodically unmaps memory pages and then traps page faults when the pages are accessed. By doing this, ANB can know the relation between the tasks and the accessed memory and determine if it should move memory closer to the tasks that reference it. Since migrating memory pages is quite expensive, ANB may moves tasks closer to the memory they are accessing instead. The adaptation of ANB to be XPV aware consists mostly in replacing the default topology information provided by the OS by the actual topology information provided by the hypervisor. Some page fault metrics used by ANB may need to be recalculated when the topology get changed. However, in order to avoid the complexity and also the fact that we don't know how long a topology will last, we decide to not recalculate the metrics in our implementation. Instead, ANB will just continue to work with the newer topology and the metrics are recalculated in a very short time.

4.3.3 Application-level modifications: the HotSpot Java virtual machine use case

We consider HotSpot 8, which uses the parallel garbage collector (GC) by default. GC divides the memory heap into two regions (usually referred as generations): a young generation and an old generation. New objects are placed in the young generation. If they survive long enough, they will be promoted and moved to the old generation. The young generation partitions its address spaces into N group spaces with N is the number of NUMA nodes. Each group space gets memory pages from the corresponding NUMA node. In order to know how to allocate memory for a given thread, HotSpot also keeps track of the CPU-to-node mapping. Regarding the old generation, it has memory pages allocated in a round-robin way over the NUMA nodes.

The modification we introduce in HotSpot to make it XPV aware is as follows. At the end of a collection, HotSpot examines the topology version counter (introduced in the system library, see Section 4.2.2) to check if the NUMA topology is stale. In case of stale topology, HotSpot creates the new group spaces and/or removes the invalid ones according to the new topology and updates the CPU-to-node mapping.

4.4 Evaluations

The previous section presented XPV, a way to virtualize NUMA while taking into account topology changes that may impact VMs. This previous section reported components that should be modified and the corresponding number of LOC required to implement XPV in different legacy systems. This corresponds to the qualitative evaluation of XPV. The current section presents the quantitative evaluation.

4.4.1 Experimental setup

The evaluations are realized on a DELL server having 8 nodes (6 AMD Opteron 6344 cores per node), each linked to a 8GB local memory. Otherwise specified, the used hypervisor is Xen and guest VMs use Linux. One node (6 cores and 8GB) is dedicated to the privileged VM (called dom0 in Xen jargon). Thus, user VMs (called domU in Xen jargon) can only use the remaining 7 NUMA nodes (42 cores and 56GB). Otherwise specified, every user VM has 20 vCPUs and 30GB memory. Note that this configuration is used because we want the VM occupies at least 4 physical NUMA nodes which, we think, are big enough to show the NUMA effects.

We experimented two application categories: those which can use an SRL (Java, C/C++) and those without an SRL (Fortran). For Java, we evaluate SpecJBB 2005 [13] (noted JBB2005) single JVM (the performance metric is the number of *business operations per second* (bops)) and BigBench [54] (the performance metric is the execution time). These applications use HotSpot as the SRL. Regarding C/C++ applications, they are evaluated with milc and lammps from Spec MPI 2007 [14]. We selected these benchmarks because they are representative of the two categories of Spec MPI applications: medium memory usage and large memory usage. These applications use TCMalloc as the SRL. We also experiment WebServing from CloudSuite [50], which uses the two SRLs and performs a lot of I/O. WebServing is a traditional web service application with four tiers: a web server, a database server, a memcached server and a client. The first three tiers are deployed in the evaluated VM while the client is deployed on a distinct server. The performance metric is the number of operations per second (ops/sec). Otherwise indicated, HotSpot and TCMalloc are launched while enabling their NUMA optimizations. Applications from the second category are bt331, fma3d, swim, mgrid331, applu331 from Spec OMP

2012 [22] and pop2 from Spec MPI 2007.

In addition to the static vNUMA solution, which exposes the topology to the VM, we also compared XPV with four different blackbox solutions. The latter are implemented within the hypervisor with the VM seeing a UMA topology. These solutions are:

- First-touch (noted FT): this solution is the basic Linux's NUMA management solution. It has been implemented within the hypervisor by Voron et al. [103].
- Automatic NUMA Balancing (noted ANB) [58]: It is the most advanced Linux solution for handling NUMA. KVM naturally includes ANB while VMware implements a similar solution. We implemented ANB in Xen for the purpose of this work. We did this by reproducing the Linux implementation of ANB in Xen. Such a proactive solution could be seen as the ideal XPV competitor.
- Interleaved: the VM's resources are packed on the minimum number of NUMA nodes and the memory is interleaved by pages of 1GB (Xen's default policy).
- No policy: the VM's vCPUs and memory are equally distributed over its allocated nodes, no NUMA policy is used.

To plot the results of all applications on the same figure using the same referential, results are normalized, over the "No policy" solution. For all plots, higher is better.

4.4.2 XPV implementation efficiency

Recall that XPV follows the same approach as vNUMA, which is the presentation of the NUMA topology to VMs. However, the implementation of XPV follows a different strategy, which requires few modifications and make it adaptable (theoretically thus far). One could ask if XPV is at least as efficient as vNUMA when the VM topology stays unchanged. To answer this question, we compared the performance of several applications when they run atop vNUMA and XPV. Recall that the implementation of XPV does not use any Xen's vNUMA code. Fig. 4.5 presents the evaluation results, interpreted as follows.

XPV performs similarly to vNUMA only with JBB2005 and pop2. However, it outperforms vNUMA for the remaining applications, by up to 75% in the case of mgrid331, and even for I/O applications (by up to 15% with webserving). This performance gap between the two solutions, which follow the same static vNUMA approach in this experiment, is explained by the fact that we were able to optimize XPV in comparison with Xen's vNUMA implementation. Several comments were posted in *Xen-dev* mailing list [48] (a scheduling issue inside the guest), underlying the inefficient implementation of vNUMA.

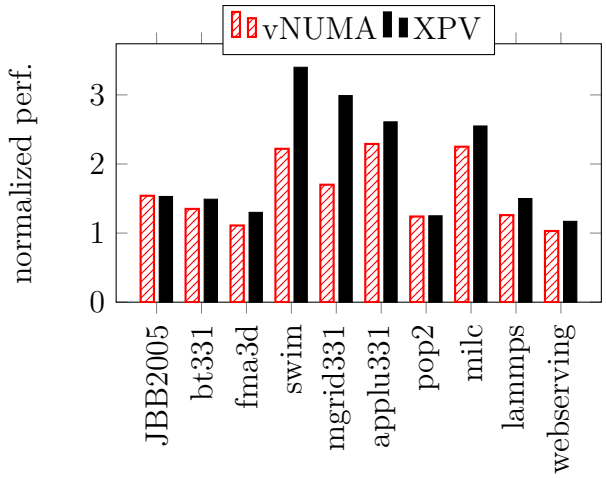

Figure 4.5: XPV implementation efficiency. XPV is compared with vNUMA (implementer by Xen) when no topology change occurs. The performance gap comes from the fact that the implementation of vNUMA by Xen is less optimized than XPV. Notice that the latter does not use any vNUMA code.

We also performed the same experiments using smaller VMs (4GB of memory with 4vCPUs). Fig. 4.6 presents the obtained results. We can observe that XPV outperforms vNUMA with most of the benchmarks (by up to 40% with swim), but the difference is smaller than with bigger VMs (53% with swim for big VMs). Therefore, for the remaining evaluations in this chapter, instead of using Xen's vNUMA as the static vNUMA baseline, we will use a static version of XPV. Thus, in the rest of this evaluation section, vNUMA refers to a static version of XPV.

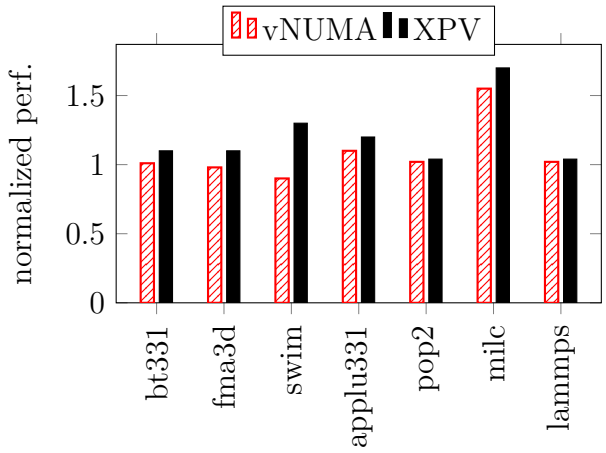

Figure 4.6: XPV implementation efficiency with small VMs.

4.4.3 vNUMA vs blackbox solutions

In this section, we compare vNUMA with the existing state-of-the-art blackbox solutions described above. During these experiments, no topology change is triggered by the hypervisor. Every application runs in a VM which uses three NUMA nodes. Fig. 4.7 presents the evaluation results, interpreted as follows.

(1) vNUMA (see "vNUMA with SRL NUMA" bars) outperforms all blackbox solutions. For instance, in the case of swim, vNUMA outperforms Interleaved, FT, and ANB by about 130%, 99%, and 88% respectively. Concerning JBB2005, which is the only benchmark used by Xen [12] and VMware [23] developers for evaluating the benefits of vNUMA, we can observe a performance gap of about 12% (which is significant) between vNUMA and other solutions. This is compliant with the results obtained by both Xen and VMware developers. However, as mentioned above, the evaluation of other benchmarks shows that vNUMA can bring much more benefits. The main reason which explains the efficiency of the vNUMA approach (the exposition of the topology to the VM) is the fact that it allows optimized NUMA policies implemented by the guest OS and the SRLs (HotSpot and TCMalloc here) to be effective.

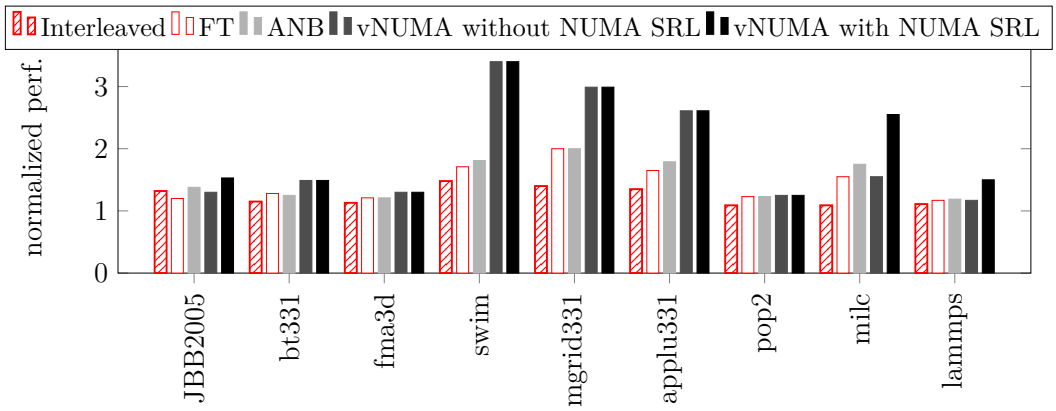

Figure 4.7: vNUMA compared with the state-of-the-art blackbox solutions (higher is better). This experiment also highlight the importance of NUMA policies embedded within the SRL. No topology change is triggered during this experiment.

(2) For some applications, the benefits of vNUMA is magnified by the optimized NUMA policies embedded within the SRL (compare "vNUMA without NUMA SRL" with "vNUMA with NUMA SRL" bars). This is the case for JBB2005, milc and lammps. For instance, milc performs 64% better when TCMalloc (its SRL) is NUMA aware atop vNUMA. Notice that applications for which "vNUMA without NUMA SRL" equals "vNUMA with NUMA SRL" do not use an SRL.

(3) One could imagine that by implementing ANB (which is a Linux NUMA solution) at the hypervisor level, it could provide the same results as using it in a NUMA VM. Our results show that this is not true (compare "ANB" with "vNUMA without NUMA SRL" bars). vNUMA outperforms ANB by up to 88% in the case of swim. This is because in the hypervisor, ANB works at the vCPU granularity, which is not as fine-grained as the thread granularity inside the guest OS. In fact, what a vCPU accesses may suddenly change if the guest schedules another task on it. Therefore, the decision to move a vCPU or a set of memory pages for minimizing remote memory accesses is not precise as it is dictated by several tasks, which is not the case when ANB runs in the guest.

(4) Among hypervisor level solutions, ANB is the best one.

4.4.4 XPV facing topology changes

This section presents the evaluation of XPV when resource management decisions taken by the hypervisor lead to NUMA topology changes for the VMs. Recall that the topology-changing decisions could be: vCPU load balancing, memory ballooning, memory flipping, and VM live migration. Since memory ballooning and memory flipping lead to the same consequences[5] (which is memory remapping), we only present the evaluation results of one of them (memory ballooning). Concerning topology changes due to VM live migration, they can be seen as a combination of the other topology change types. Therefore, our evaluations focus on topology changes caused by vCPU load balancing and memory ballooning.

To show the benefits of each XPV feature, we evaluated two XPV versions:

- XPV with topology change notifications confined within the guest OS. The SRL level is not informed. This means that only NUMA policies implemented within the guest kernel are aware of topology changes. This version is noted "OS only XPV".
- XPV with topology change notifications taken into account by both the guest kernel and the SRL. This version is simply noted "XPV".

We compared these two versions with blackbox solutions and a static version of XPV noted vNUMA as stated above. We present and discuss in this section only the results for three representative applications since the other results that we have observed are similar: a representative Java application (JBB2005), a representative C application (milc), and a representative application which does not use an SRL (swim).

[5] We experimented flipping and observed that negative impact for a memory intensive app which runs inside the VM which is subject to flipping.

4.4.4.1 XPV facing topology changes caused by vCPU loadbalancing

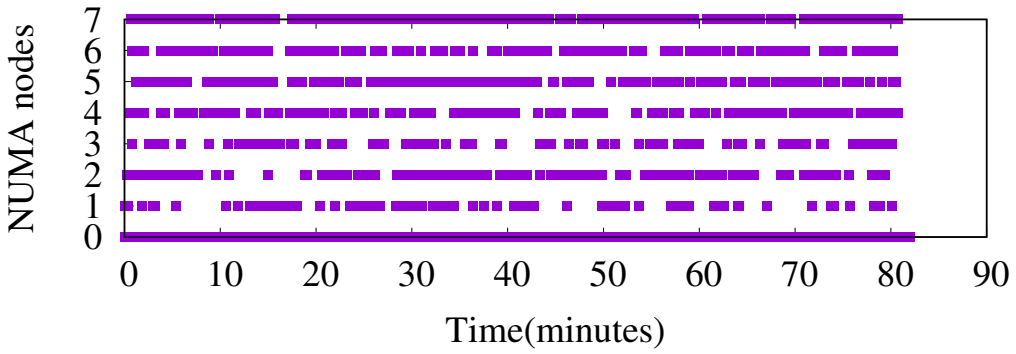

Figure 4.8: NUMA nodes occupied by $vCPU_0$ of JBB2005 VM during its lifetime when overcommitment is done on the CPU resource.

Recall that vCPU loadbalancing could lead to vCPU migrations between different nodes, thus changing the topology of VMs. To create a situation which may lead the hypervisor performing vCPU loadbalancing, we run each tested application in three identical VMs (started at the same time). The total number of vCPUs (for the three VMs) is 48 while the number of available cores is 42 (recall that 6 cores are dedicated to the dom0). Thus, the scheduler of the hypervisor is likely to realize vCPU loadbalancing during the experiment. As an example, Fig. 4.8 shows the different nodes occupied by $vCPU_0$ of JBB2005 VM during its execution. For this application, we counted 1695 topology changes during the execution, corresponding to about 20 topology changes per minutes (noted TC/min). The topology change rates for milc and swim are 5 TC/min and 12 TC/min respectively.

Fig. 4.9 presents the performance of each application when different NUMA virtualization solutions are used. These results are interpreted as follows. (1) We can see that JBB2005 does not suffer a lot from the issue of topology changes due to vCPU migrations. Except FT, the performance gap between XPV and other solutions (about 12%) is almost the same as when there is no topology change (presented in Fig. 4.7). FT provides the lowest performance, 58% lower than XPV. In fact, FT only considers NUMA for the first memory allocation operations. Interleaved suffer less because it has interleaved the VM's memory, thus increasing the probability for a vCPU to access a local memory. Concerning ANB, it enforces memory locality by relocating either vCPUs or memory chunks. (2) Things are different concerning milc and swim. The performance gap between XPV and other solutions is higher in this case

(90% and 55% on average for milc and swim respectively). This means that milc and swim are sensitive to topology changes caused by vCPU loadbalancing. (3) The static vNUMA solution significantly degrades the application performance when topology changes are triggered. The performance gap with the adaptable XPV solution is about 12%, 127% and 84% in the case of JBB2005, milc and swim respectively. (4) Making topology changes visible to the SRL layer improves the performance of some applications. The gap between "OS only XPV" and XPV is about 12% for JBB2005 and 65% for milc (recall that swim does not use an SRL). (5) XPV keeps all applications almost to their best performance, which is the one observed when no topology change is triggered (presented in Fig. 4.7).

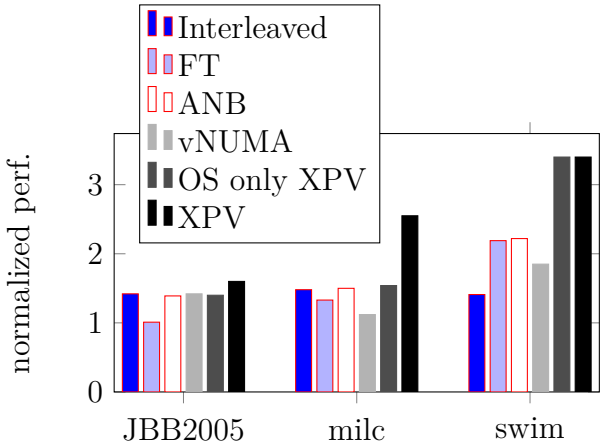

Figure 4.9: XPV facing vCPU loadbalancing due to the overcommitment of the CPU resource (higher is better).

4.4.4.2 XPV facing topology changes caused by memory ballooning

To realize this experiment, we used Badis, a memory overcommitment system presented in [85]. Badis is able to dynamically adjust the memory size of VMs which share the same host in order to give to each VM the exact amount of memory it needs. We ran Badis and our three applications (JBB2005, swim and milc) at the same time. Each VM is launched with 10vCPUs and 20GB which can be adjusted by Badis. There is no CPU overcommitment in this experiment.

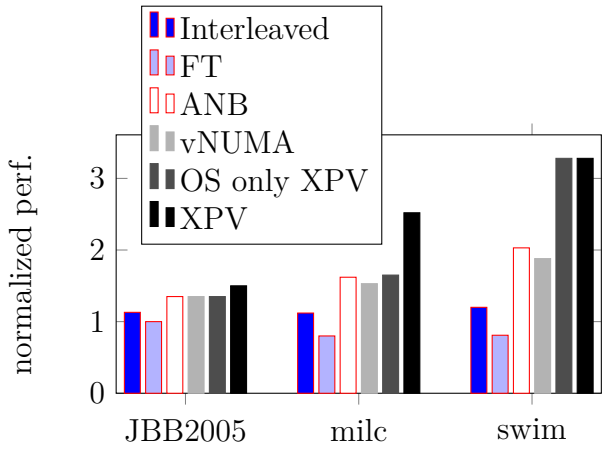

Figure 4.10: XPV facing memory ballooning (higher is better).

Fig. 4.10 presents the evaluation results. The interpretation of these results is almost the same as the one presented in the previous section. The only difference with the previous experiments if the fact that both Interleaved and FT provide very bad performance. For instance, in the case of swim, XPV outperforms Interleaved and FT by about 173% and 304% respectively.

4.4.5 Automatic NUMA Balancing (ANB) limitations

Due to the inability of existing vNUMA solutions to handle VM topology changes, ANB like solutions have been envisioned as the best compromise thus far. In the performance point of view, our evaluation results confirmed that ANB is the best blackbox solution, although it is largely outperformed by XPV. However, ANB has two side effects which can degrade the performance of the hypervisor. First, ANB decisions can enter in conflict with resource management decisions performed by the hypervisor. For instance, a vCPU loadbalancing decision can move a vCPU to a node, resulting in remote memory accesses, thus the intervention of ANB. The latter will move back the vCPU to its source node, thereby contradicting the previous resource management decision. This issue has also been identified by VMware [23]. Second, a malicious VM can manipulate ANB as follows. Let us consider a VM booted with two vCPUs, one vCPU per node. Let us consider its memory distributed on the two nodes. Even if the VM is presented a UMA topology (as ANB does), an application inside the VM can dynamically discover the distance between vCPUs an a memory chunk using read/write latencies (the STREAM benchmark [80] is the perfect candidate). Therefore, a malicious application can enforce

remote memory accesses in order to force ANB in the hypervisor to continually migrate the corresponding VM's vCPUs. These migrations would lead to the migration of tenant VMs, thus impacting their execution. We implemented such a malicious VM and validated this issue.

4.4.6 XPV internals

To evaluate the low level overhead introduced by XPV, we evaluated XPV internal mechanisms. The latter are:

1. the new interrupt handler in the OS used each time there is a topology update,
2. the new syscall used by the SRL to check the topology version and to retrieve the topology information from the OS (for instance in our experiments, it is called each time the GC runs),
3. the search for a free page on a particular node,
4. the update of memory allocator's data structures (per-CPU caches and the central buddy allocator).

First, the new interrupt handler in XPV runs in about 368 CPU cycles, which is negligible. Second, the new syscalls added to the GC, we found that the GC execution time is not significantly impacted. We rely on virtual dynamic shared object (vDSO) [33], which is a mechanism provided by the kernel for exporting some frequently used read-only syscalls to user-space applications. vDSO routines are called as regular routines, without worrying about performance overhead. Thus, the time consumed by the new syscall is negligible. Third, about the search for a free page, it takes a negligible time to do so as all pages in the kernel are indexed by node (you can refer to Section 4.3.2). Finally, concerning the update of memory allocator's data structures, whenever a page is freed and returns to the allocator, the kernel examines on which node the page resides and puts it in the correct location. This hardly has any impact on application performance.

4.5 Related Work

Several work investigated the problem of efficiently handling NUMA architectures in virtualized environments. Most of them were implemented by hypervisor providers (Xen, VMware, Hyper-V), and, to the best of our knowledge, only six academic works investigated this issue.

4.5.1 Industrial solutions

Xen [78]. Xen tries to pack the VM's resources on a single node, called the home node. When the VM requires more than one node, Xen proposes both Interleaved and static vNUMA. As shown in this chapter, none of these solutions are efficient as XPV.

Oracle VM Server [27]. Oracle VM server proposes the policies used by Xen. It suffers thus the same limitations.

VMWare [23]. VMWare works like Xen, but does not allocate the memory with an interleaved policy. Instead, the VM's memory is simply spread over nodes. Concerning its static vNUMA based solution, VMware is able to update the virtual NUMA topology of the VM by changing the ACPI tables. However, its effectiveness depends on the capability of the guest OS to adapt itself on topology changes. This is not currently the case in the mainstream OSs, which makes the VMware's solution inefficient. In this chapter, our XPV solution is able to dynamically adapt the NUMA policies of both the OS and the SRLs when the hypervisor changes the topology.

KVM [9]. KVM implements two blackbox solutions through Linux. These solutions are First-touch (FT) and Automatic NUMA Balancing (ANB). FT is inefficient with SRL based applications while ANB has a lot of limitations as presented in the previous section (conflict with resource management decisions taken by the hypervisor and vulnerable in the point of view of security).

Hyper-V [25]. Hyper-V uses the static vNUMA approach. When it creates a VM, Hyper-V exposes the bootstrap NUMA topology to the VM. Hyper-V does not handle the issues related to topology changes [51].

4.5.2 Academic solutions

To the best of our knowledge, all academic solutions use a blackbox approach, meaning that the VM sees a UMA topology and the hypervisor implements the NUMA policy. Disco [34] is a hypervisor that enforces locality on a NUMA machine by using page migration and page replication. When the hypervisor observes that a page is intensively used remotely, it migrates or replicates the page. Disco hides the NUMA topology, which makes this solution inefficient for an SRL that implements its own NUMA policy. Similarly, Rao et al. [91], Wu et al. [107], Jaeung al. [61] and Liu et al. [77] proposed new heuristics to place or to migrate the memory or the vCPUs in order to efficiently use the NUMA architecture. However, in their work, they hide the NUMA topology to the VM, which also makes them inefficient for many SRLs. All these solutions are similar to ANB, which has been discussed above.

Instead of proposing new NUMA policies, Voron et al. [103] proposed to implement NUMA policies used in Linux and the Carrefour policy [53] in the hypervisor. The VM can then choose the most efficient NUMA policy. Because a single NUMA policy can not be efficient for all applications, letting an application selects its policy is better than using a single fixed policy. However, the solution proposed by Voron et al. only considers memory (thread placement is not studied). The solution also assumes that all applications inside a VM use the same NUMA policy, which is not the best strategy if a VM runs several processes. Moreover, the solution hides the NUMA topology to the VM, which makes it inefficient for many SRLs. Finally, the solution requires a lot of engineering efforts, while we show that XPV only requires a modest engineering effort.

4.5.3 Positioning of our work

As shown in previous sections, all existing solutions fail to virtualize NUMA efficiently. With XPV, we propose to make the static vNUMA approach dynamic. XPV exposes to the guest OS and its SRLs the exact actual NUMA topology. By doing so, XPV allows each layer in the virtualization stack to do what it does best: resource utilization optimization for the hypervisor and NUMA management for SRLs, helped by the guest OS.

4.6 Synthesis

In this chapter we presented XPV, a new principle for virtualizing NUMA. XPV adopts an opposite approach in comparison with existing solutions. In fact, instead of managing NUMA at the hypervisor level, XPV presents to the VM its actual topology while tracking topology changes. We presented a systematic way to integrate in less than 2k LOC XPV in two legacy hypervisors (Xen and KVM), two legacy guest OSs (Linux and FreeBSD), and three system runtime libraries (HotSpot, TCMalloc, and jemalloc). We evaluated XPV with different Java and C benchmarks. The evaluation results showed that XPV outperforms all existing solutions, by up to 304%.

Chapter 5

Conclusion and Perspectives

5.1 Conclusion

With the emergence of cloud computing, researches on virtualization have become very active among computer science researchers. At its core, virtualization is about introducing a hardware abstraction layer between the OS and the physical hardware. Virtualization techniques for CPU, memory and standard I/O devices are well established. Virtualization of micro-architectural components such as processor caches or NUMA still have rooms for improvements. The main difficulties are not only due to the fact that there is no proper way to physically partition these components in the current hardware architectures but also that the hypervisor can reconfigure the resource placement for a VM at will, at any time without the guest OS knowing as a result of load balancing. We showed that these factors can bring severe performance impacts to VMs sharing the same underlying physical hardware. It's worth pointing out that consolidating a maximum number of VMs on a minimum number of physical machines is a common practice in cloud computing as it can increase hardware utilization rates and decrease power consumption in the data centers.

In this thesis, we proposed software solutions called Kyoto and XPV to virtualize processor caches and NUMA machines respectively. With Kyoto, the main idea is to associate and enforce a LLC pollution permit to a VM. The pollution permit is defined as a function of LLC cache misses and enforced by the vCPU scheduler in the hypervisor. We implemented Kyoto in three popular virtualization systems: Xen, KVM and Pisces. The results from our experiments validate Kyoto's effectiveness in terms of performance isolation

and predictability. They also show that Kyoto introduces a negligible overhead. Henceforth, using the Kyoto system, the cloud provider may compel cloud users to book pollution permits for their VMs.

Through experiments, we showed that exposing to a VM its actual NUMA topology is the best way to handle NUMA in virtualized systems. To virtualize a NUMA machine, XPV extends the well known paravirtualization principle in two directions. First, in the same way as paravirtualization actually abstracts away I/O devices, XPV extends the principle by also abstracting away the physical NUMA topology into a virtual NUMA topology that can change at runtime. Second, while currently paravirtualization is only used at the guest OS level to implement optimized drivers for virtualized environments, XPV extends this principle to the SRLs: we propose to adapt the SRLs with paravirtualization techniques in order to dynamically adapt their NUMA policy when the virtual NUMA topology changes. Thanks to XPV, several VMs can run on a NUMA machine and still use NUMA policies efficiently.

5.2 Perspectives

In Kyoto, we focused mainly on the LLC contention because it is one of the most well-known and critical types of resource contention. We would like to investigate further into other types of contention such as contention for the memory controller and for network resources. Memory controller contention occurs when several processors and I/O devices request for memory access at the same time. A careful study about the memory controller's behavior and strategies for handling simultaneous requests can potentially reveal some optimization opportunities to minimize the contention. Besides, the problem of contention, caused due to the share of network resources by VMs or applications is also an interesting topic. For instance, user VMs usually have access to network disk servers. To improve the performance, a SSD cache layer can be placed between the user VMs and the storage servers. Normally, the SSD cache layer is implicitly shared among the user VMs, leading to contention.

Regarding XPV, we would like to apply the principle to other types of non-uniform architectures such as Non-Uniform Input/Output Access (NUIOA) where I/O devices can connect to specific CPUs making them access to some memory nodes faster than to the others. In case of NUIOA, I/O performance depends on the placement of processes on CPUs. We think that exposing

NUIOA topology to the VM allows the latter to achieve the best I/O performance possible in a virtualized environment.

NUMA architecture is an evolution from the traditional multiprocessor shared memory architecture as an attempt to provide scalable memory bandwidth. However, a NUMA machine is still a tightly coupled computing package: a workload can only use resources available on a single machine and there is no easy way for a machine to share its spare resources. In addition, it remains impossible to integrate new CPU or memory technologies into such machine. The only option is to replace the whole machine with a new one. To avoid this terrible waste and to improve resource utilization, many hardware vendors have proposed the so-called disaggregated architecture. Disaggregation means separating computing resources (processors, memory) and allowing each type of resource to be assigned independently to workloads. The communication between different resources can be done through high-performance network technologies such as Infiniband, Photonic Interconnect and iSCSI. One of the benefits of a disaggregated architecture is the ability to add or remove individual processor or memory module. A disaggregated architecture is generally implemented at the rack scale. As a result, such an architecture is easier to manage in a datacenter, and the topology is more uniform than that of a NUMA architecture. However, modules may be heterogeneous, leading to performance unpredictability. It would be interesting to investigate virtualization support in such architectures.

Bibliography

[1] The hotspot group. http://openjdk.java.net/groups/hotspot/. Visited on June 2020.

[2] Hyper-v vnuma enable. https://specs.openstack.org/openstack/nova-specs/specs/ocata/implemented/hyper-v-vnuma-enable.html. Visited on June 2020.

[3] jemalloc memory allocator. http://jemalloc.net/. Visited on June 2020.

[4] Kyoto protocol. http://unfccc.int/kyoto_protocol/items/2830.php. Visited on June 2020.

[5] Lmbench - tools for performance analysis. http://www.bitmover.com/lmbench/. Visited on June 2020.

[6] Memory flipping in kvm. https://wiki.osdev.org/Virtio. Visited on June 2020.

[7] Spec cpu 2006. https://www.spec.org/cpu2006/. Visited on June 2020.

[8] Understanding full virtualization, paravirtualization, and hardware assist. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf. Visited on June 2020.

[9] What is linux memory policy? https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt. Visited on June 2020.

[10] What is numa. https://www.kernel.org/doc/html/latest/vm/numa.html. Visited on June 2020.

[11] What is numa. https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html. Visited on June 2020.

[12] Xen on numa machines. https://wiki.xen.org/wiki/Xen_on_NUMA_Machines. Visited on September 2018.

[13] Specjbb2005 (java server benchmark). https://www.spec.org/jbb2005/, 2005. Visited on June 2020.

[14] Spec mpi 2007. https://www.spec.org/mpi2007/, 2007. Visited on June 2020.

[15] Amd-v nested paging. Tech. rep., AMD, Inc, 2008. Visited on June 2020.

[16] Pci-sig single root i/o virtualization (sr-iov) support in intel virtualization technology for connectivity. https://www.intel.com/content/dam/doc/white-paper/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.pdf, 2008. Visited on June 2020.

[17] Understanding memory resource management in vmware esx server. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/perf-vsphere-memory_management.pdf, 2009. Visited on June 2020.

[18] 5 lessons we've learned using aws. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html, 2010. Visited on June 2020.

[19] Pci-sig sr-iov primer: An introduction to sr-iov technology, 2011.

[20] Enabling optimized interrupt/apic virtualization in kvm. https://www.linux-kvm.org/images/7/70/2012-forum-nakajima_apicv.pdf, 2012. Visited on June 2020.

[21] Intel virtualization technology for directed i/o (vt-d): Enhancing intel platforms for efficient virtualization of i/o devices. https://software.intel.com/content/www/us/en/develop/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices.html, 2012. Visited on June 2020.

[22] Spec omp 2007. https://www.spec.org/omp2012/, 2012. Visited on June 2020.

[23] The cpu scheduler in vmware vsphere 5.1. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-cpu-sched-performance-white-paper.pdf, 2013. Visited on April 2018.

[24] Amd i/o virtualization technology (iommu) specification. http://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf, 2016. Visited on June 2020.

[25] Deploying virtual numa for vmm. https://technet.microsoft.com/en-us/library/jj628164.aspx, 2016. Visited on June 2020.

[26] Intel 64 and ia-32 architectures developer's manual: Vol. 3a. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2016. Visited on June 2020.

[27] Optimizing oracle vm server for x86 performance. http://www.oracle.com/technetwork/server-storage/vm/ovm-performance-2995164.pdf, 2017. Visited on June 2020.

[28] Ackaouy, E. The xen credit cpu scheduler. http://www-archive.xenproject.org/files/summit_3/sched.pdf, 2006. Visited on June 2020.

[29] Ahn, J., Kim, C., Han, J., Choi, Y.-R., and Huh, J. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing* (USA, 2012), HotCloud'12, USENIX Association, p. 19.

[30] Ahn, J. H., Li, S., O, S., and Jouppi, N. P. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *ISPASS* (2013), IEEE Computer Society, pp. 74–85.

[31] Apparao, P., Iyer, R., and Newell, D. Implications of cache asymmetry on server consolidation performance. In *2008 IEEE International Symposium on Workload Characterization* (Seattle, WA, USA, Oct. 2008), IEEE, pp. 24–32.

[32] 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. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03, ACM, pp. 164–177.

[33] Bovet, D. P. Implementing virtual system calls. https://lwn.net/Articles/615809/, 2014. Visited on September 2018.

[34] Bugnion, E., Devine, S., and Rosenblum, M. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (1997), SOSP '97, ACM, pp. 143–156.

[35] Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst. 30*, 4 (Nov. 2012).

[36] Bugnion, E., Nieh, J., and Tsafrir, D. *Hardware and software support for virtualization.* No. # 38 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Rafael, CA. OCLC: 1014037684.

[37] Bungale, P. P., and Luk, C.-K. Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (New York, NY, USA, 2007), VEE '07, Association for Computing Machinery, p. 137–147.

[38] Cherkasova, L., and Gardner, R. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005), ATEC '05, USENIX Association, pp. 24–24.

[39] Cherkasova, L., Gupta, D., and Vahdat, A. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev. 35*, 2 (Sept. 2007), 42–51.

[40] Chisnall, D. *The Definitive Guide to the Xen Hypervisor. Chapter: Understanding How Xen Approaches Device Drivers.*, first ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[41] Denneman, F. Decoupling of cores per socket from virtual numa topology in vsphere 6.5. http://frankdenneman.nl/2016/12/12/decoupling-cores-per-socket-virtual-numa-topology-vsphere-6-5/, 2016. Visited on June 2020.

[42] Denneman, F. Impact of cpu hot add on numa scheduling. http://frankdenneman.nl/2017/04/14/impact-cpu-hot-add-numa-scheduling/, 2017. Visited on July 2018.

[43] Dhiman, G., Marchetti, G., and Rosing, T. Vgreen: A system for energy-efficient management of virtual machines. *ACM Trans. Des. Autom. Electron. Syst. 16*, 1 (Nov. 2010).

[44] Drepper, U. *What every programmer should know about memory;*. 2007.

[45] Du, J., Sehrawat, N., and Zwaenepoel, W. Performance profiling in a virtualized environment. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (USA, 2010), HotCloud'10, USENIX Association, p. 2.

[46] Du, J., Sehrawat, N., and Zwaenepoel, W. Performance profiling of virtual machines. *SIGPLAN Not. 46*, 7 (Mar. 2011), 3–14.

[47] Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., and Veidenbaum, A. V. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (USA, 2012), MICRO-45, IEEE Computer Society, p. 389–400.

[48] Faggioli, D. Pv-vnuma issue: topology is misinterpreted by the guest. https://lists.xenproject.org/archives/html/xen-devel/2015-07/msg03241.html, 2015. Visited on June 2020.

[49] Fedorova, A., Seltzer, M., and Smith, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (USA, 2007), PACT '07, IEEE Computer Society, p. 25–38.

[50] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, ACM, pp. 37–48.

[51] Finn, A. Hyper-v dynamic memory versus virtual numa. https://www.petri.com/hyper-v-dynamic-memory-versus-virtual-numa, 2015. Visited on June 2020.

[52] Gaud, F., Lepers, B., Decouchant, J., Funston, J., Fedorova, A., and Quéma, V. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (2014), USENIX ATC'14, USENIX Association, pp. 231–242.

[53] Gaud, F., Lepers, B., Funston, J., Dashti, M., Fedorova, A., Quéma, V., Lachaize, R., and Roth, M. Challenges of memory management on modern numa systems. *Communications of the ACM 58*, 12 (Nov. 2015), 59–66.

[54] Ghazal, A., Ivanov, T., Kostamaa, P., Crolotte, A., Voong, R., Al-Kateb, M., Ghazal, W., and Zicari, R. V. Bigbench v2: The new and improved bigbench. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (April 2017), pp. 1225–1236.

[55] Gidra, L., Thomas, G., Sopena, J., and Shapiro, M. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, ACM, pp. 229–240.

[56] Gidra, L., Thomas, G., Sopena, J., Shapiro, M., and Nguyen, N. Numagic: A garbage collector for big data on big numa machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 661–673.

[57] Goldberg, R. P. Survey of virtual machine research. *Computer 7*, 9 (Sept. 1974), 34–45.

[58] Gorman, M. Automatic numa balancing. https://lwn.net/Articles/523065/, 2012. Visited on September 2018.

[59] Gupta, A., Kale, L. V., Milojicic, D., Faraboschi, P., and Balle, S. M. Hpc-aware vm placement in infrastructure clouds. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering* (USA, 2013), IC2E '13, IEEE Computer Society, p. 11–20.

[60] Hagimont, D., Mayap Kamga, C., Broto, L., Tchana, A., and De Palma, N. Dvfs aware cpu credit enforcement in a virtualized system. In *Middleware 2013* (Berlin, Heidelberg, 2013), D. Eyers and K. Schwan, Eds., Springer Berlin Heidelberg, pp. 123–142.

[61] 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 *Proceedings of the 2010 Conference on Parallel Processing* (2011), Euro-Par 2010, Springer-Verlag, pp. 615–623.

[62] Heechul Yun, Gang Yao, Pellizzoni, R., Caccamo, M., and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (Philadelphia, PA, Apr. 2013), IEEE, pp. 55–64.

[63] Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., and Reinhardt, S. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, Association for Computing Machinery, p. 25–36.

[64] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, S., and Emer, J. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, Association for Computing Machinery, p. 208–219.

[65] Jerger, N. E., Vantrease, D., and Lipasti, M. An Evaluation of Server Consolidation Workloads for Multi-Core Designs. In *2007 IEEE 10th International Symposium on Workload Characterization* (Boston, MA, USA, Sept. 2007), IEEE, pp. 47–56.

[66] Jha, M. Whats new in vsphere 6.0 - vnuma enhancements. https://alexhunt86.wordpress.com/2015/05/16/whats-new-in-vsphere-6-0-vnuma-enhancements/, 2015. Visited on June 2020.

[67] Jin, X., Chen, H., Wang, X., Wang, Z., Wen, X., Luo, Y., and Li, X. A Simple Cache Partitioning Approach in a Virtualized Environment. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications* (Chengdu, Sichuan, China, 2009), IEEE, pp. 519–524.

[68] Kaminski, P. Numa aware heap memory manager. https://developer.amd.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf, 2012. Visited on June 2020.

[69] Kang, K.-D., Alian, M., Kim, D., Huh, J., and Kim, N. S. VIP: Virtual Performance-State for Efficient Power Management of Virtual Machines. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad CA USA, Oct. 2018), ACM, pp. 237–248.

[70] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07* (2007).

[71] Klee, D. Vmware vsphere 6.5 breaks your sql server vnuma settings. https://www.davidklee.net/2016/11/29/vmware-vsphere-6-5-breaks-your-sql-server-vnuma-settings/, 2016. Visited on June 2020.

[72] Knowlton, K. C. A fast storage allocator. *Communications of the ACM 8*, 10 (Oct. 1965), 623–624.

[73] Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z., and Pu, C. An Analysis of Performance Interference Effects in Virtual Environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software* (San Jose, CA, USA, Apr. 2007), IEEE, pp. 200–209.

[74] Lapata, M. Automatic evaluation of information ordering: Kendall's tau. *Comput. Linguist. 32*, 4 (Dec. 2006), 471–484.

[75] 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* (2015), USENIX ATC '15, USENIX Association, pp. 277–289.

[76] Liu, M., Li, C., and Li, T. Understanding the Impact of vCPU Scheduling on DVFS-Based Power Management in Virtualized Cloud Environment. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems* (Paris, France, Sept. 2014), IEEE, pp. 295–304.

[77] 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)* (Minneapolis, MN, USA, June 2014), IEEE, pp. 325–336.

[78] Liu, W., and Ufimtseva, E. vnuma in xen. https://events.static.linuxfound.org/sites/events/files/slides/vNUMA%20in%20Xen_XenDev.pdf, 2014. Visited on June 2020.

[79] Mars, J., and Soffa, M. L. Synthesizing contention. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, NY, USA, 2009), WBIA '09, Association for Computing Machinery, p. 17–25.

[80] McCalpin, J. Stream: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report.

[81] Milojicic, D. High performance computing (hpc) in the cloud. *Computing Now* (9 2012).

[82] Nathuji, R., and Schwan, K. Virtualpower: Coordinated power management in virtualized enterprise systems. In *In Proceedings of International Symposium on Operating System Principles (SOSP* (2007).

[83] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10*, 03 (aug 2006).

[84] Nikolaev, R., and Back, G. Perfctr-xen: A framework for performance counter virtualization. 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. 15–26.

[85] Nitu, V., Kocharyan, A., Yaya, H., Tchana, A., Hagimont, D., and Astsatryan, H. Working set size estimation techniques in virtualized environments: One size does not fit all. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 2*, 1 (Apr. 2018), 19:1–19:22.

[86] Ouyang, J., Kocoloski, B., Lange, J. R., and Pedretti, K. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, Association for Computing Machinery, p. 149–160.

[87] Paul, I., Yalamanchili, S., and John, L. K. Performance impact of virtual machine placement in a datacenter. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)* (Austin, TX, USA, Dec. 2012), IEEE, pp. 424–431.

[88] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (July 1974), 412–421.

[89] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, Association for Computing Machinery, p. 381–391.

[90] Qureshi, M. K., and Patt, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (USA, 2006), MICRO 39, IEEE Computer Society, p. 423–432.

[91] 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)* (Feb 2013), pp. 306–317.

[92] Riesen, R., Brightwell, R., Bridges, P. G., Hudson, T., Maccabe, A. B., Widener, P. M., and Ferreira, K. Designing and implementing lightweight kernels for capability computing. *Concurr. Comput. : Pract. Exper. 21*, 6 (Apr. 2009), 793–817.

[93] Robin, J. S., and Irvine, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9* (USA, 2000), SSYM'00, USENIX Association, p. 10.

[94] Rosenblum, M., Bugnion, E., Devine, S., and Herrod, S. A. Using the simos machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul. 7*, 1 (Jan. 1997), 78–103.

[95] Ruprecht, A., Jones, D., Shiraev, D., Harmon, G., Spivak, M., Krebs, M., Baker-Harvey, M., and Sanderson, T. Vm live migration at scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2018), VEE '18, ACM, pp. 45–56.

[96] Simons, J. vnuma: What it is and why it matters. https://octo.vmware.com/vnuma-what-it-is-and-why-it-matters/, 2011. Visited on June 2020.

[97] Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. Binary translation. *Commun. ACM 36*, 2 (Feb. 1993), 69–81.

[98] Tang, L., Mars, J., and Soffa, M. L. Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (New York, NY, USA, 2011), EXADAPT '11, Association for Computing Machinery, p. 12–21.

[99] Tang, L., Mars, J., Vachharajani, N., Hundt, R., and Soffa, M. L. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA '11, Association for Computing Machinery, p. 283–294.

[100] 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* (2016), EuroSys '16, ACM, pp. 3:1–3:14.

[101] Teabe, B., Tchana, A., and Hagimont, D. Billing system CPU time on individual VM. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (Cartagena, Colombia, May 2016), IEEE, pp. 493–496.

[102] Teimouri, D. Numa and vnuma - back to the basic. https://www.teimouri.net/numa-vnuma-back-basic/#.WuSuy9axXCJ, 2017. Visited on June 2020.

[103] 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* (2017), EuroSys '17, ACM, pp. 453–467.

[104] Wang, X., Wen, X., Li, Y., Luo, Y., Li, X., and Wang, Z. A Dynamic Cache Partitioning Mechanism under Virtualization Environment. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications* (Liverpool, United Kingdom, June 2012), IEEE, pp. 1907–1911.

[105] Watson, J. Virtualbox: Bits and bytes masquerading as machines. *Linux J. 2008*, 166 (Feb. 2008).

[106] West, R., Zaroo, P., Waldspurger, C. A., and Zhang, X. Online cache modeling for commodity multicore processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, Association for Computing Machinery, p. 563–564.

[107] Wu, S., Sun, H., Zhou, L., Gan, Q., and Jin, H. vprobe: Scheduling virtual machines on numa systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept 2016), pp. 70–79.

[108] Xie, Y., and Loh, G. H. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, Association for Computing Machinery, p. 174–183.

[109] Zeng, L., Wang, Y., Shi, W., and Feng, D. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *Proceedings of the 2013 International Conference on Cloud Computing and Big Data* (USA, 2013), CLOUDCOM-ASIA '13, IEEE Computer Society, p. 267–274.

[110] Zhang, X., Dwarkadas, S., and Shen, K. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, Association for Computing Machinery, p. 89–102.

[111] Zhuravlev, S., Blagodurov, S., and Fedorova, A. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, Association for Computing Machinery, p. 129–142.