SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

École doctorale EDITE de Paris (ED130)
Informatique, Télécommunication et Électronique

Thèse de doctorat de Sorbonne Université

Spécialité : **Ingénierie / Systèmes Informatiques**

Présentée par : **Gauthier Voron**

Pour obtenir le grade de :
Docteur de Sorbonne Université

Sujet de la thèse :
Virtualisation efficace d'architectures NUMA

Soutenue le : **8 mars 2018**
Devant le jury composé de :

Rapporteurs :	R. Guerraoui	Professeur, École Polytechnique Fédérale de Lausanne
	L. Réveillère	Professeur, Université de Bordeaux
Examinateurs :	V. Quéma	Professeur, Université Grenoble-Alpes
	E. Encrenaz	Maître de conférences, Sorbonne Université
	P. Sens	Professeur, Sorbonne Université
	G. Thomas	Professeur, Télécom SudParis

LIP6

Contents

Contents **ii**

1 Introduction **1**

2 State of the Art **7**
- 2.1 Multicore architectures . . . 8
  - Architecture details . . . 9
  - Related challenges . . . 13
- 2.2 Non Uniform Memory Access architectures . . . 20
  - Architecture details . . . 21
  - Linux NUMA policies . . . 26
  - Related challenges . . . 27
- 2.3 System virtualization . . . 34
  - Technical details . . . 35
  - Architecture details . . . 38
  - Related challenges . . . 41
- 2.4 Software settings . . . 47
- 2.5 Conclusion . . . 48

3 The Well-Known Bottlenecks **49**
- 3.1 The Virtualized I/O Overhead . . . 49
  - Hardware Emulation . . . 49
  - The I/O Memory Management Unit . . . 50
  - Evaluation of IOMMU . . . 53
- 3.2 The virtualized IPI overhead . . . 54
  - Usage of the IPI . . . 54
  - Implementations of IPI . . . 55
  - The libactive library . . . 57
  - Evaluation of vIPI . . . 58
- 3.3 The Xen load balancer . . . 59
  - Completely Fair Scheduler . . . 60
  - Credit Scheduler . . . 60
  - The libpin . . . 60
  - Evaluation of pining scheme . . . 62

3.4 Conclusion 64

4 The NUMA Bottleneck **65**

4.1 NUMA policies under study 65

Comparison of NUMA policies effects 66

The NUMA policy selection metric 66

4.2 Current hypervisor NUMA management 69

The default round-1G policy 69

Huge pages and splinterring 70

Evaluation of the default policy 71

4.3 Conclusion 72

5 Virtualization of NUMA Architectures **73**

5.1 Improved hypervisor NUMA management 74

The Xen implementation of round-4K 75

The Xen implementation of Carrefour 75

Limitations of PEBS with virtualization 79

The Xen implementation of first-touch 79

Limitations of first-touch with IOMMU 82

The Xen NUMA policy selection interface 82

5.2 Evaluation 83

Evaluation of vNUMA on a single machine 83

Evaluation of vNUMA on several machines 87

5.3 Conclusion 88

6 Conclusion **91**

6.1 Future works 92

6.2 Perspectives 93

Bibliography **95**

CHAPTER 1

INTRODUCTION

In 1965, Gordon E. Moore publishes a short paper, named "Cramming More Components onto Integrated Circuits" [50], containing one of the best-known conjectures in computer science: "The number of components per integrated circuit will double every 18 months". Until the early 2000s, this conjecture nonetheless was verified, but also had the effect, as a fortunate consequence, to rise up the clocks frequency. In these years, simply waiting for the next generation hardware was sufficient to increase the software performance. However, physical constraints, such as thermal dissipation difficulties, prevent CPU manufacturers to keep increasing the clock frequency. Thus, until this year (2018), the Moore conjecture is still verified since the number of transistors in a chip actually keeps doubling. The manufacturers then use these additional components to provide more than a computing unit per chip, naming them *cores*[1]. As a result, it is common nowadays to have a dozen cores in a single personal computer.

While the increasing number of cores in a chip makes it able to execute more instructions in a given time, these instructions have to be executed in parallel. This constraint drastically changes the conditions for software to gain performance on new hardware. A programmer now needs to design its algorithms to be parallelizable so when the number of core increases, the amount of instruction executed in parallel increases too. Parallelizing a program is, however, a difficult task as it generally implies to add synchronization mechanisms for the different cores to coordinate with each other.

Furthermore, increasing the number of core worsens an already existing problem. As each core needs to access the memory, either to fetch instructions or to manipulate data, adding more cores puts more pressure on the memory system. This issue was already known before the 2000s as the memory system clock was not as fast as the cpu clock, and then unable

[1]In this document, we use terms *CPU* and *core* interchangeably as they both designate some circuitry able to execute a sequence of instructions.

to satisfy every CPU request. The hardware designers used to add fast memory caches to absorb the cost of the memory accesses. With more than a dozen cores, however, even this cache mechanism is not sufficient to contain the memory load and the memory system saturates. This gap between the cores needing more information to work and a memory system unable to scale up to serve the cores requests led to a major modification in the computer's architecture. As the processor has been redesigned to contain multiple cores when it became impossible to improve a single processing unit performance, the memory system now contains multiple memory banks and controllers as it became impossible to speed up a single memory unit. Because of this distributed aspect of the memory, the access time to data is not uniform anymore and depends on where it is placed on the computer. This architecture is thence called Non-Uniform Memory Access (NUMA) architecture. As a result, the performance of a process is today constrained by the parallelization of its algorithms but also by a good dispatch of its data.

In parallel to the evolution of hardware architecture, the hardware usage also changes with the rise of cloud computing. Indeed, as the hardware becomes more powerful, it executes more complex tasks more quickly. Nevertheless, this hardware power and complexity are nothing but a waste of processing power if there is no consuming task to execute. As this waste of electricity also means a proportional waste of money, a common practice is to share the same hardware between multiple users to maximize the computer usage. This way of using computers is not particularly new, the UNIX operating system already was multiuser. Nowadays, however, this tendency is so widespread that tens of completely unrelated users can share the same hardware somewhere in a data center owned by a cloud provider. As the users generally have different needs, such as what operating system they want to use, the cloud providers usually emulate several virtual machines on their powerful hardware and rent them to users. With this technology, called virtualization, the same hardware can be shared between several users where each of them has the illusion to control its own computer with the freedom to configure it to suit its needs.

The virtualization technology and the NUMA architecture both evolved independently to tackle different issues: reduce hardware usage cost for the first, produce more powerful hardware for the second. Nonetheless, nowadays, the hardware used in the cloud data centers uses NUMA architectures and thus, the virtual machines are executed atop such hardware. The virtualization software has, however, not been designed for NUMA architectures. Because of this poor integration, the applications executed inside a virtual machine running atop of a NUMA architecture may have low performance. As the combined use of NUMA architectures and virtualization is relatively recent, because of the cloud computing emergence, only a few works [56, 44] address this performance issue.

My PhD thesis addresses the challenge of efficiently virtualizing a NUMA architecture in a cloud infrastructure. In detail, my research is twofold. On the first side, my research has the goal of measuring how virtualization behaves on a NUMA architecture, and how and why a NUMA architecture changes the performance of virtualized applications. On the other side, my PhD thesis proposes new techniques to efficiently use NUMA architectures when the hardware is virtualized. The contributions of this research are:

- A systematic performance analysis of a large set of applications from different benchmarks suites on a large NUMA virtualized system.
- For each already known performance bottleneck due to virtualization, the implementation of a solution showing that the overhead is not affected by the NUMA architecture.
- An evaluation of the already known NUMA memory management algorithms in operating systems and the proposition of a metric to choose which algorithm provides the best performance.
- The adaptation of the main NUMA memory management algorithms in operating systems to a virtualized system, an implementation of them on the Xen hypervisor and an evaluation of this implementation with a representative set of 29 applications.

Each contribution is explained in the following chapters of this document. These contributions, as well as the experimental evaluations lead to the following conclusions:

- With a single virtual machine spanning 48 cores, our systematic performance analysis shows that, in the set of applications we choose, the overhead is mainly caused by the I/O virtualization, the inter-processor interrupts (IPI) virtualization, and a bad NUMA memory management by the hypervisor.
- While the I/O and IPI virtualization causes important performance losses, the existing approaches to solve these issues give good results, even on a 8 nodes NUMA hardware. Additionally, a third supposed bottleneck, the thread and virtual CPU load balancing only causes marginal performance losses on a NUMA architecture.
- The metric we propose allows the user to choose the best NUMA memory management algorithm by executing the application once. This algorithm is nonetheless the best one for a native system but is also the best option in most cases for a virtualized system.
- The adaptation and evaluation of the main NUMA memory management algorithms in operating systems to virtualized systems show that the current Xen hypervisor NUMA management is highly inefficient. More precisely, with a single virtual machine spanning 48 cores, using an efficient NUMA policy divides the completion time of 9 applications by more than 2, with an observed maximum improvement of 6 times. With consolidated workloads of multiple virtual machines, among the 11 tested configurations, using an efficient NUMA policy reduces the completion time of at least one virtual machine by more than 2 in 9 cases.

The remaining of this document is organized in 5 chapters. Chapter 2 describes the state of the art for NUMA architectures and system virtualization, Chapter 3 focuses on the already known virtualization bottlenecks, Chapter 4 focuses on the NUMA architecture effects on

performance for both native and virtualized systems, Chapter 5 describes our solution to implement a NUMA management in the Xen hypervisor as well as an evaluation of this solution and Chapter 6 concludes the thesis. More precisely:

State of the Art
Chapter 2 describes the existing work related to NUMA architectures and system virtualization with the aim to show that on one side, achieving good performance on a NUMA architecture requires explicit communication and resource management, while on the other side, the virtualization, by design, hides hardware and software aspects, thus preventing explicit approaches. Prior to focusing on NUMA architecture details, the first part describes the general multicore architecture and the main mechanisms for several cores to synchronize. This part also presents some previous works that aim to improve multicore performance and shows that a way to achieve that is to not use the synchronization mechanisms provided by the hardware and instead make the synchronization explicit in the software. The second part describes some key characteristics of the NUMA architecture as well as some well-known NUMA memory management policies. This part also describes related works that aim to improve performance by exploiting the knowledge of the NUMA topology. The last part briefly presents the different virtualization techniques and gives additional details about the techniques we focus on in this research, especially about memory management. This part also presents some previous works addressing the issues caused by virtualization.

The Well-Known Bottlenecks
Chapter 3 evaluates three possible performance bottlenecks for virtualized applications. The first bottleneck is the I/O virtualization which is described at an architectural level in the first part. We show that we can remove this bottleneck by using a dedicated part of the hardware implemented in modern computers. In the second part, we describe how this hardware part works and interacts with the memory management circuitry. In the third part, we evaluate the effects of this hardware part on the applications performance. The second bottleneck is the inter-processor interrupts virtualization. We first describe how the IPI mechanism is implemented in a native system, then in a virtualized one and show what causes the overhead. We then describe a solution to remove the vIPI overhead, based on previous works. We finally evaluate the performance of applications when using this solution. The third supposed bottleneck is the load balancer. A first part describes how the hypervisor and guest operating system load balancers work. A second part describes a tool we use to evaluate the impact of the load balancer on application performance by constraining the decisions it can take. A third part evaluates the impact of the load balancer decisions on the applications performance.

The NUMA Bottlenecks
Chapter 4 focuses on the impact of the NUMA architecture on application performance in both native and virtualized systems. A first part evaluates the importance of the NUMA memory management on application performance on a native system. This is done by comparing the effects of the NUMA memory management algorithms described in Chapter 2 on application execution times. This part also describes the NUMA policy selection

metric and evaluates its efficiency. A second part describes the NUMA memory management algorithm used in the Xen hypervisor. We also describe what effects this algorithm has on the memory accesses of the virtual machine. This algorithm is then evaluated by comparing the applications performance executed in Xen and in a native system.

Virtualization of NUMA Architectures
Chapter 5 describes how we implement an efficient NUMA management in the hypervisor. In the first part, we show how we adapt each of the NUMA memory management algorithms described in Chapter 2 to a virtualized configuration. We also outline the limitations we encountered while adapting these NUMA policies and how a cloud user could use them through a policy selection interface. Then, the second part evaluates the performance gains of the applications when using these NUMA memory management algorithms in a virtualized system. We evaluate both single virtual machine configurations and consolidated virtual machine configurations.

Conclusion
Chapter 6 concludes the thesis with a summary of the lessons learnt about the virtualization of NUMA architectures. The thesis shows that combining the system virtualization technology with the NUMA architecture raises resource management issues. When the architecture becomes complex, like with NUMA hardware, the software layer that manages the hardware resources has to know the hardware details so it can use it efficiently. The system virtualization technology isolates the guest operating system from the hardware, preventing it from taking efficient decisions. We show that we can overcome this issue by shifting the NUMA management from the guest operating system to the hypervisor. Additionally we show, with an implementation in the Xen hypervisor and with an evaluation over a large set of applications, that existing NUMA management techniques are as effective in the hypervisor as in an operating system. Moreover, we show that a hypervisor that uses the appropriate NUMA policies executes NUMA virtual machines with no overhead. This chapter also discusses future work and perspective. The future work that we propose focuses on three axes: (i) increase the performance of already implemented NUMA policies, (ii) implement additional NUMA policies and (iii) compare our approach to address the virtual NUMA management to other possible approaches. In the perspective part, we look at this thesis from a larger point of view and discuss the implications of this work and of the lessons we learnt.

CHAPTER 2

STATE OF THE ART

My PhD thesis addresses the challenge of efficiently virtualizing a NUMA architecture. This chapter presents the existing works in both NUMA architecture and system virtualization fields with the following goals:

- Describe the concepts related to NUMA architectures and explain how to reach good performance on this hardware. Specifically, this chapter shows that the more the end user program knows about the NUMA topology and the hardware details, the more it can use the machine efficiently.

- Describe the concepts related to system virtualization and especially to the virtualization of memory. In particular, this chapter shows that, by design, nowadays virtualization isolates the virtualized system from the hardware and hides the topology details.

Achieving good performance on a NUMA hardware requires an explicit access to the hardware information that, precisely, the system virtualization hides to the guest system. This contradiction makes the efficient virtualization of NUMA architectures a challenging task.

As an introduction to NUMA architectures, this chapter presents the general multicore architectures. Indeed, the NUMA architecture is a special kind of multicore architecture. It is then necessary to understand how an application can be efficient on a multicore hardware. The first part of this chapter thus focus on the specificities of multicore architectures and how a poor awareness of these characteristics can lead to scalability issues. This first part also presents existing works that address these performance issues.

Interestingly, this chapter shows that, as for NUMA architectures, the more an application knows about the multicore hardware details, the more it can scale. Moreover, the third

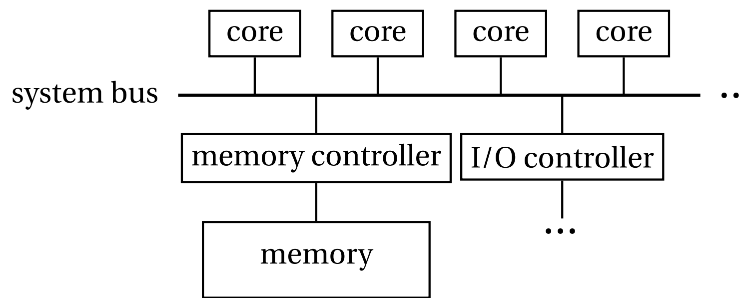

Figure 2.1: The architectural view of an multicore architecture

part of this chapter, dedicated to system virtualization, shows that existing works related to multicore virtualization address issues caused by the opacity enforced by the virtualization. This opacity prevents the virtualized software to use the hardware details to scale. The parallel we can make between multicore and NUMA virtualization tends to show that the virtualization opacity causes issues that require important research efforts.

2.1 Multicore architectures

As stated in Chapter 1, in order to use an increasing number of transistors, modern processors are built from several computation units called *cores*, each executing a different instruction sequence. In this section, we focus on multicores without considering the NUMA architectures. We consider thus a small multicore, in which the cores, the devices and the memory are all connected to a single bus. Figure 2.1 illustrates this architecture. This (relatively simplified) architecture is currently used in desktops, laptops and smartphones when the processor provides several cores.

On this architecture, the main performance concern for a program is to efficiently exploit all the available cores. To this end, a common method is to split the program algorithm in smaller and independent tasks. Each of these tasks are executed on different cores. It is, however, necessary to collect the results of these tasks and to merge them in order to obtain the final result. Moreover, it may be impossible to split an algorithm into completely independant tasks. Instead, some computations can only be decomposed in mostly independant tasks that must occasionally synchronize. Since these synchronization steps may limit the scalability of applications, we study them in this section.

This section first presents the synchronization mechanisms provided by modern multicore architectures, namely the MOESI cache coherency protocol, and the inter-processor interrupts. For each of these mechanisms, this section describes its purpose, how it works and what performance issue it can cause when used without caution. Some existing works focusing on the performance improvement on multicore architecture are then presented with a specific emphasis on how the described solutions use, or avoid using, the implicit multicore synchronization mechanisms.

Architecture details

The hardware provides several synchronization mechanisms, but two of them are especially important in the context of my study. The first mechanism is the MOESI cache coherency protocol that allows all the cores of a system to have the same view of the memory, even if the main memory data is outdated with respect to some core local cache. This first mechanism is implicit since the program has no explicit control on this coherency, except to not work on the same data from different cores. The second mechanism is the inter-processor interrupt: the way for cores to asynchronously notify other cores that an event occurred. As opposite to the MOESI protocol, the inter-processor interrupt is an explicit mechanism since the communication aspect, as a synchronization mechanism is ultimately a form of communication, is explicit.

The MOESI protocol

As a processor consumes instructions much faster than the memory can serve data[1], chip manufacturers include a cache hierarchy between the processing unit and the memory controller. When the processor modifies the data, it stores them in the cache and data are eventually, at an unpredictable time, committed to memory. During the time the modified data are in the cache but not yet committed to the main memory, the main memory stores an outdated state of the data. On a monocore architecture, this is not an issue since the only accessing core asks the cache controller for the data before to ask memory. The unique core of the system is thus sure to fetch the most up-to-date state of the data. On a multicore architecture, each core has its own cache hierarchy, at least for the first levels of cache. Each of these caches is able to store modified states of data and commit them to the main memory, possibly in a different order than the writes occurred. This breaks the sequential consistency property of the memory.

To overcome the resulting consistency issues, the hardware implements a cache coherency protocol. When a cache controller is asked for a data item, it ensures that the state it has of this data item is the most up to date state known by any cache controller in the system. To do so, modern cache controllers implement a distributed reader-writer lock algorithm. Several versions of this algorithm exist, but they all follow these two principles:

- Several cache controllers can hold the same data item in parallel as lon as none of them try to modify the data item. When asked by a core to read a data item, a cache controller thus probes the other cache controllers for the sought data item. The other controllers can share it without breaking the sequential consistency.
- Only a single cache controller can modify a data item at a given time. When asked by a core to modify a data item, a cache controller must ensure that none of the other controllers hold the data item. To this end, this cache controller sends invalidation requests to the other controllers for the modified data item. The other controllers can later probe for this data item and fetch the modified state.

[1] On today hardware, a single core can consume data 100 times faster than the main memory can serve it.

	Not last writter	Last writter
Exclusive owner	Exclusive	Modified
Sharing owner	Shared	Owned
Not owner	Invalid	

Table 2.1: MOESI States

A cache controller conforming to these principles ensures that it always provides the most recent version of a data known in the memory system. By doing so, the cache controllers ensure the memory is sequentially consistent.

In addition to preserve the memory sequential consistency, the cache controllers try to minimize the number of communications. Indeed, accessing to the cache is a frequent operation for a core. Hence, broadcasting to every cache controller of the system each time a core asks for a data item would be inefficient and not scalable. A common way to provide a reader-writer lock semantic with a low number of required communcation is the MOESI procotol. This protocol defines a set of possible states a data item can have. The state of a data item is relative to a cache controller. Different cache controllers in the system may thus associate different states to the same data item. The state of a data indicates to the cache controller if the data can be read or written, or if the cache controller must communicate with the other controllers first. The MOESI protocol defines five states. One of them, the Modified state represents the read-write acquisition of the lock. Three other ones, the Exclusive, Shared and Owned states represent the read-only acquisition of the lock. The last one is the Invalid state that represents the not acquired lock.

From a communication point of view, the Invalid state indicates that the data item is not present in the local cache and that it is necessary to probe all the other caches, then if none of them holds the latest state of the data item, fetch it from main memory. The four remaining states indicates (i) if the local cache holds the unique copy of the data item outside of the main memory and (ii) if the local cache is the last one that has modified the data item. The first indication is useful to avoid communications: if the local cache is the only one to work on a data item, there is no need to probe the other caches when reading or writing the data item. The indication of what is the last modifier of a data item is used to know if the data item must be committed from the local cache to the main memory if the cache is flushed. Table 2.1 presents the MOESI states and what they indicate.

Excepted if a data is Invalid, a cache controller can always read it without changing its state. Reading a data with an Invalid state leads the cache controller to probe the other caches. If one other cache contains the data, it is fetched with the Shared state, otherwise it is read from the main memory and has the Exclusive state. Since the local cache does not modify the data, both the Exclusive and the Shared state indicate the local cache is not the last writer. Modifying a data from any state puts it in the Modified state, which effectively indicates the local cache is the last modifier. The Modified state also indicates the local cache is the exclusive owner. Indeed, when a cache modifies a data, the other copies become outdated and are invalidated by a probe. The probes are not sent if the data was in the

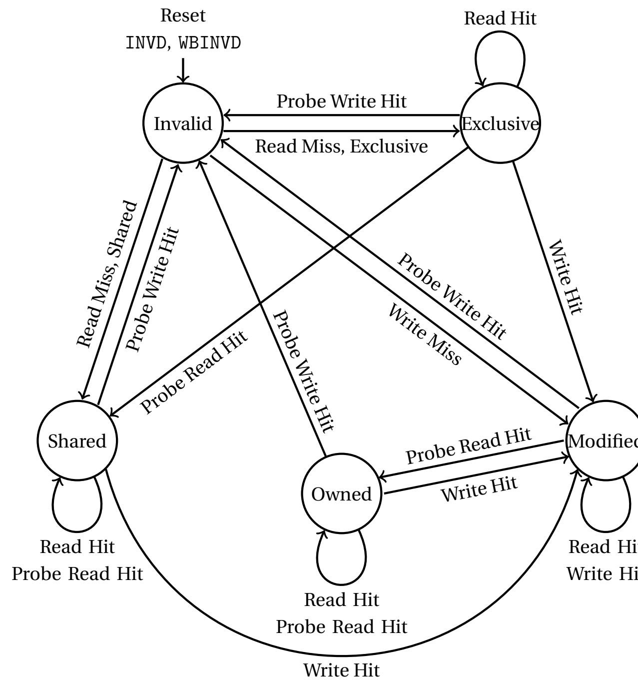

Figure 2.2: MOESI State Transitions

Exclusive state since no copy could exist in this case. If a cache owns a data in the Exclusive state and another cache probes to read this data, it goes in the Shared state. If a cache owns a data in the Modified state and another cache probes to read this data, it goes in the Owned state. The Owned state indicates that the data is shared between several caches but the last writer on this data is local cache. The local cache is then responsible to commit the data to main memory if the cache is flushed. Beside this, there is no difference with the Shared state. Figure 2.2 summarizes the transitions between the MOESI states.

By conforming to the MOESI protocol, the cache controllers hide the complexity of hardware to the cores. Whereas the memory system is composed of the main memory and of several caches, the cache controllers expose a flat topology to the cores. While this simplifies the task of programmers, ignoring the communications a memory access can cause may lead to produce highly inefficient applications. These are typical examples of hidden communication inefficiencies.

- False sharing: If two or more variables are packed in a small address range, they may be considered by the cache controllers as a single data. If these variables are modified by different cores, the data keeps moving from a cache to another, causing a lot of costly communications.

- Migration cost: It is frequent that several tasks update successively the same data structure. If different cores execute these tasks, even one by one, the data has to

migrate from one cache to another, again causing costly communications.

- Ticket lock: A simple implementation of a spinlock is the ticket lock where several cores spin reading the same variable. This variable is updated at each lock release. If the lock is highly contended, each release causes an invalidation for all the waiters, which then broadcast probe requests. If the lock is frequently released, it causes a massive amount of useless communications.

The downside of the cache coherency protocol is that, by hiding the communcation caused by memory accesses, it generates a high number of messages on the system bus and makes the cores spend a lot of time stalled, waiting for cache controllers answers. As presented in Section 2.2, this phenomenon worsens on a NUMA architecture where the number of cores is higher and even the main memory is distributed. The issues related to the MOESI protocol is also a good example of the performance overhead an application may encounter if it does not take the hardware details into account, which is the case for virtualized software as we explain in Section 2.3.

Inter-processor interrupts

The MOESI protocol and the cache coherency mechanisms in general allow the core to communicate by exchanging data. Synchronization, however, requires that a core signals an event to one or many other cores. This can be done by using only memory, as for the spinlocks. This method, however, has two drawbacks. First, a waiting core has to spin on a memory location and to continuously pull a possible update issued by the signaling core. This constant pulling causes a waste of CPU cycles and, despite the modern processors energy saving instructions, a waste of energy. Then, since a processor can only pull with this method, it is impossible for a core to asynchronously notify another core of an unexpected event.

In order to provide a push notification mechanism, modern processors provide the notion of interrupt. Conceptually, an interrupt is a message emitted by a device and received by one or several cores. When the core receives the message, it reacts by executing some code defined by the system prior to the reception. When the core has executed this code, it continues its previous execution. This mechanism can be used by I/O devices to signal that an I/O operation occured, for instance, a disk controller has finished transfering a disk sector into main memory. Additionally, on modern hardware, a core can stop fetching new instructions and halt. Once halted, a core only consumes a very low amount of energy. The reception of an interrupt wakes a core up and allows it to fetch instructions again.

A special kind of interrupt is the inter-processor interrupt (IPI). A core can issue an IPI to notify another core that some system defined event should be handled. To indicate precisely the nature of the event, it is common for the sending core to store information in a well known memory location that the receiving core can read upon receiving the interrupt. In addition, the sending core sometimes needs to be sure the target core has handled the event before to continue its execution. In this case the sending core can use a pull method, as spinning on an acknowledgement memory location.

On the architecture we use, the IPI mechanism goes through a special device called the Advanced Programmable Interrupt Controller (APIC). There is one APIC per core, included in the processor chip. Sending an IPI is thus an extremely fast operation since it does not require to acquire the system bus or any circuitry shared by the other cores. For security reasons, however, it is common for the operating system to restrict the APIC usage to the kernel only.

The inter-processor interrupts are thus an explicit communication mechanism. As opposite to the cache coherency mechanism, the system programmer is fully aware of every transmitted message. However, since the access to the APIC, and thus the ability to send IPI, is restricted to the kernel code, application programmers cannot use this mechanism explicitely. Additionally, the kernel programmers assume that sending an IPI is a fast, almost instantaneous, operation and then use it in critical path code, such as synchronization primitives provided to applications. Section 3.2 shows that sending an IPI can be a slow operation in a virtualized system and explains how this incorrect assumption reduces the application performance.

Synthesis

In this part, we show that modern multicore hardware provides several mechanisms for the different cores to communicate and synchronize. Some of these mechanisms, like the cache coherency MOESI protocol, are implicit: they hide hardware complexity to the end programmer by presenting a simple hardware with more convenient properties, like sequential consistency. Some of these mechanisms, like the IPI, are explicit: they allow the programmer to deal with the hardware complexity and to exploit it. These two kinds of mechanisms have a cost. The only difference is that the programmer is most likely to have this cost in mind when dealing with explicit mechanisms than with implicit ones.

Related challenges

In the previous section, we describe several communication mechanisms provided by multicore architectures. The study of these mechanisms is an introduction to the specificities of the NUMA architectures. In this section, we present works focusing on how to exploit or deliberately avoid exploiting these communication mechanisms in order to improve performance. These works only address inter-core communication issues. Especially, this section does not present works addressing cache pollution issues or other multithread related issues, as these are not specific to multicore architectures.

Table 2.2 enumerates the works we present. For each of them, we present the issue they address and the approach they use. We can classify the addressed issues in two categories: (i) contention issues where too many cores access the same resource at the same time, which cause access serialization, (ii) hardware related issues where there is no contention but a naive usage of the hardware leads to hidden expensive operations. We can also characterize the used approach by looking the amount of inter-core communication it induces:

Solution	Issue	Approach	Exposure
Remix	Hardware	MOESI awareness	Hidden
Scalloc	Contention	MOESI awareness + Private data space	Hidden
Sloppy Counter	Hardware	Private data space + Lazy merging	Hidden
The Multikernel	Hardware	Private data space	Explicit
The Factored OS	Hardware	Share nothing	Explicit
The Scalable Commutativity Rule	Contention	No communication	Explicit

Table 2.2: Works related to multicore architectures

MOESI awareness These approaches use the hardware efficiently, making sure to not generate unnecessary communications between the cache controllers. They consider worst-case scenarios where a lot of cache invalidation occurs but could be avoided, for instance, with a different memory layout. Except for these specific worst cases, the application still relies on cache coherency.

Private data space The different cores only work on their own data located in a memory zone they access exclusively. By avoiding to share data, the cores avoid to synchronize by cache coherency messages. The software can however use the memory as an explicit communication channel between cores. Several data spaces can be merged on demand. Communicating by memory is efficient but requires that a cache coherency protocol is implemented by hardware. Additionally, communicating by memory is not well suited for asynchronous notifications.

Share nothing The cores cannot share memory. The cores then do not exchange cache coherency messages. The cores can, however, communicate by explicit channels, like IPI or I/O. Such approaches do not require a cache coherency protocol to be implemented by hardware. It is then appropriate to be used in large scale systems with hundreds of cores and even in distributed systems.

No communication The system is designed so there is no need for the cores to communicate. This approach is the most scalable but also the most compelling. Indeed, using this approach may require to change the design of the application at a high level.

In addition to the *Issue* and *Approach* criteria, Table 2.2 indicates in the last column if the presented work hides the hardware details to the end user application or if it exposes them.

Remix

In a multicore system, several cores sharing data need to synchronize. This synchronization causes inter-core communication and often worsens the application performance. By contrast, if each core works on its own data, there is no need to communicate. In modern processors, the cache controllers work on data item of fixed size, called cache line. If two

or more variables are packed in an address range smaller than a cache line size, they are considered by the cache controllers as a single data. Two or more cores can thus work on unrelated variables but still force cache controllers to communicate. This problem is called *false sharing*.

In optimized native applications, experimented programmers can ensure there is no false sharing by choosing wisely the memory layout of their variables. In a managed runtime environment, like a Java Virtual Machine, the memory layout is chosen by the environment. It is then difficult for the programmer to fix false sharing issues.

Eizenberg et al. [32] address this problem by modifying the JVM. At runtime, the JVM uses hardware sampling counters to detect where false sharing occurs. The sampling counters capture, for each core, which memory addresses are set in the Modified state. If two or more addresses are set in the Modified state by different cores in the same cache line, the JVM considers the cache line is falsely shared.

Once the JVM has detected which cache line is falsely shared, it repairs it. The JVM first identifies the objects containing the problematic cache lines. Then, the JVM stops the mutator threads and for each problematic object, modifies the memory layout of all the objects of the same class. Technically, the JVM adds padding between the fields accessed by several cores to place them on different cache lines. The JVM finally resumes the mutator threads.

This work thus addresses a hardware related issue: the inability of the cache controllers to manage data smaller than a cache line size. The approach is to change the memory layout to match the implicit memory partitioning made by the cache controller. This solution, however, is invisible to the end user application.

Scalloc

The false sharing is a special case of data sharing. When several cores access the same variable to modify it, it is called *true sharing* or simply *sharing*. Usually, the application protects this shared data with locks. These locks force to serialize the accesses, thus nullifying the performance gain of the multicore chip.

Memory allocators are typical programs that try to avoid sharing situations [4, 11, 57]. We call a memory allocator a library that provides the `malloc()` and `free()` primitives. This library is a bridge between the `mmap()` system call that allocates memory at a memory page granularity and the application which allocates memory at various granularities, from the byte to several megabytes. Memory allocators usually implement several buffers larger than a memory page that can be fragmented into smaller regions. The memory allocator returns these regions, called objects, to the application on a `malloc()`. There are different fragmentation size, or class size, for these buffers, so the allocator can allocate for several granularities. When the application releases an object, it is tagged free in its buffer and is added to the free list of the buffer.

In a multicore context, several cores may concurrently allocate objects of the same size. A naive solution would be to protect the size class buffers with locks. This solution would

result in a sharing situation, causing poor allocation performance. Instead, memory allocators usually maintain one size class buffer per application thread. These buffers are called Thread Local Allocation Buffers (TLAB). With this design, the threads do not share any buffer and allocating an object requires no lock.

However, using TLAB causes 2 issues. First, if a thread releases all the objects of a TLAB, the memory allocator should make this TLAB available to the other threads. Indeed, a memory allocator should avoid wasting memory space. Transferring a TLAB from a thread to another one is a form of sharing. Second, a thread can allocate an object, and another thread can release it. This means that using TLAB does not prevent several threads to share buffers.

Aigner et al. solve these issues with Scalloc [3]. First, Scalloc implements a buffer backend where the threads put empty TLAB. This backend is an array of lock-free stacks, with as many stacks as there are cores. Each application thread is associated to one of these stacks. The stacks are evenly associated to the threads. Each application thread releases and allocates TLAB from its associated stack. This design ensures that TLAB are shared between a relatively small number of threads. If its associated stack is free, a thread allocates a new TLAB from a larger zone called the arena. The arena is shared between all the threads but is rarely used.

When a thread releases an object it has allocated previously, it puts it in the free list of its TLAB. The thread can pop an object from this free list on a later allocation. But when a thread releases an object that another thread has allocated, using the free list of the allocating TLAB would mean to share this free list. Since the free list may be frequently used by the thread owner of the TLAB, sharing this list could cause a serialization overhead. A scalloc TLAB uses 2 free lists. The first one is the local free list, used only by the owner thread. The second one is the remote free list. If a thread releases an object from a TLAB not owned by the thread, the object goes to the remote free list of the TLAB. The remote free list is thus shared between the threads not owning the TLAB. Since releasing an object allocated by another thread is an uncommon operation, this does not affect the performance. When the owner thread allocates an object, it tries to pop one from the local free list. When the local list is empty, it looks the size of the remote free list. Only if this free list is full enough, it moves the content of this remote free list to the local one. Otherwise, the allocating thread fetches a new TLAB from the backend.

Scalloc addresses a contention problem: how to handle a large amount of threads allocating and releasing objects concurrently. The general solution is to use TLAB, which are private allocation spaces and avoid communication between threads, and thus, between cores. When it is necessary to communicate, Scalloc makes sure to reduce the amount of implied cores and that communication happens as rarely as possible. However, as an allocator, Scalloc only expose the `malloc()` and `free()` primitives without exposing the hardware details to the application.

Sloppy Counter

Memory allocators such Scalloc solve the issue of shared data structures by using several private thread local structures instead. This kind of per thread or per core replication avoids the cores to communicate and thus, improves performance.

However, some data structures are intended to be shared between cores to allow them to communicate. This is the case for the reference counter. This kind of counter is often used for reference-counted garbage collection of various resources. While a core uses a resource, it keeps a reference on this resource. Keeping a reference means that before using the resource, the core increments the associated reference couter. When the core stops to use the resource, it releases the reference by decrementing the counter. When the counter drops to zero, the resource is garbage collected. The reference counter is hence a communication protocol between the cores.

Usually, a reference counter is implemented with atomic operations. Atomic operations are special instructions provided by the processor making a read then a, possibliy conditional, write on a cache line while keeping the cache line locked. Because the processor keeps the cache line locked, it prevents other cores to access it between the read and the write. However, atomic operations are implemented atop cache coherent caches. Many cores updating the same reference counter concurrently thus generate a high amount of cache coherency messages. These communication decrease the application performance.

Boyd-Wickizer et al. [14] propose a scalable implementation of the reference counter called the Sloppy Counter. The sloppy counter is composed of a global counter and one local counter per core. In this structure, the local counter acts as a cache for the global counter. When a core takes a reference, it tries to acquire one in its local counter. If the local counter is non-zero, the core decrements it and no communication occured. If the local counter is zero, the core then increments the global counter. To release a reference, a core only increments its local counter and, once again, no communication occurs. Because references are released in the local counter, this one rarely reaches zero. Accessing the global counter is thus an uncommon operation.

The global counter and the local ones are then merged only when the system decides to perform a garbage collection. The merge operation simply consists to subtract the local counters to the global one. The result is the new value of the global counter while all the local counters are reset to zero. If the global counter is zero, then the resource is garbage collected.

The sloppy counter thus avoids unnecessary cache coherency communications between cores. To this end, it is structured with one shared space accessed rarely: the global counter, and one private space per core, accessed frequently: the local counters. The sloppy counter is an implementation of an existing mechanism, the reference counter, and hence does not expose any additional hardware details.

The Multikernel

The previous approaches systematically hide the multicore aspect of the hardware. Remix [32] silently modifies the classes memory layout, Scalloc [3] only provides the `malloc()` and `free()` primitives and the sloppy counter [14] provides `put()` and `get()` primitives.

In the problem addressed by Remix, the false sharing, the JVM is the cause: it prevents the programmer to choose an efficient memory layout, which would avoid to create false sharing. The solution is thus implemented in the JVM. There is no reason to inform the user application about the JVM decisions.

By contrast, in the problems addressed by Scalloc and the sloppy counter, the cause is the user application. The congestion occurs because too many threadd access the same resource pool or because too many cores modifies the same counter. Both Scalloc and the sloppy counter solve the problem by duplicating the implied data structures. However, the threads or cores ultimately need to communicate. Because of that, Scalloc sometimes share TLAB between threads and the sloppy counter merges its local and global parts on a garbage collection.

Hiding the multicore nature of the hardware is convenient because it makes the new solutions compatible with existing software. Nontheless, while the system hides the multicore details of the hardware to the application, this application will, at some point, require expensive communication between cores, decreasing its performance. The following works expose the hardware to the user application.

Several works [13, 40, 52] address the multicore scalability problem by redesigning operating systems. The Multikernel [9] is representative of these works. The general idea is to consider each core as a separated computer with its own operating system. Each of these operating system has its own data structures and work only on them. This is typically the case for the scheduler structures.

For other operating system functions that need a global state, like process memory management, each operating system has its own replicate of the global state. The cores keep the replicated states consistent with explicit synchronization based on message passing. The message passing is implemented with some dedicated memory zones used as a channel to transfer cache-line-sized messages point-to-point between single writer and reader cores.

The Multikernel thus avoid unnecessary communication between cores, whether cache coherency messages or inter processor interrupts. To this end, each operating system in the Multikernel only works on its own private space and perform explicit communication when needed.

The Factored Operating System

The previously presented works make communication explicit by avoiding to use shared memory. Instead they maintain private state that can be synchronized with message passing. This message passing is implemented with cache coherency.

Wentzlaff et al. [69] argue that implementing a cache coherency protocol may not even be possible in a computer with thousands of cores. Indeed, even if the software rarely use cache coherency, a local cache controller with a cache line in an Invalid state still needs to probe other cache controllers if the core requires the cache line. Thus, the cache controllers still exchange messages, even if the cores only use private data structures.

The Factored Operating System (FOS) is an operating system designed to execute in a computer without a global cache coherency. The FOS is based on an idea similar to the Multikernel [9]: execute an autonomous part of the system on each core. Unlike the Multikernel, the FOS does not maintain any global state. Instead, each core works as a server that perform one task: either execute application threads, one at a time, or execute a kernel service. Additionally, each core maintains a name cache. The name cache is a routing table that indicate what core to contact for a given service to execute. The cores communicate with message passing that is architecture dependent.

The Scalable Commutativity Rule

The works presented in this section address the communication effects on performance. The first work, namely Remix [32], remove the useless cache coherency messages. Indeed false sharing messages do not actually maintain coherency because the cores work on distinct variables. The next works, namely Scalloc [3] and the sloppy counter [14], reduce the amount of exchanged message by batching them. The cores make most of the updates on private data and commit these changes to a global state when required. The two last works, namely the Multikernel [9] and the FOS [69], reduce the amount of required messages. Because they are designed like distributed systems, they maintain private states, one per core that rarely or never have to be synchronized.

All of these presented works however require that the cores synchronize between each others at some point. Clements et al. describe a general method to design a software which does not require that the core synchronize. To be so, a software must provides an interface that follows the Scalable Commutativity Rule [22]. This rules states that, given a set S of operation executed concurrently, the result of an operation o executed after S must not depends on the ordering of operations inside S. If a software interface follows that rule, then it exists an implementation of this interface that can executes the operations concurrently with no need of synchronization.

By using this rule, a programmer can design a system which does not require any expensive communication. This actually solves contention issues by making the multicore aspect of the underlying hardware part of the software conception.

Conclusion

Previous works brings different solutions to address the problem of system efficiency and scalability in a multicore context. However, they are all built atop of at least one of these two general concept: (i) synchronization necessarily induces serialization, thus a system can be scalable only if its data is shared as little as possible between the cores, (ii) hardware

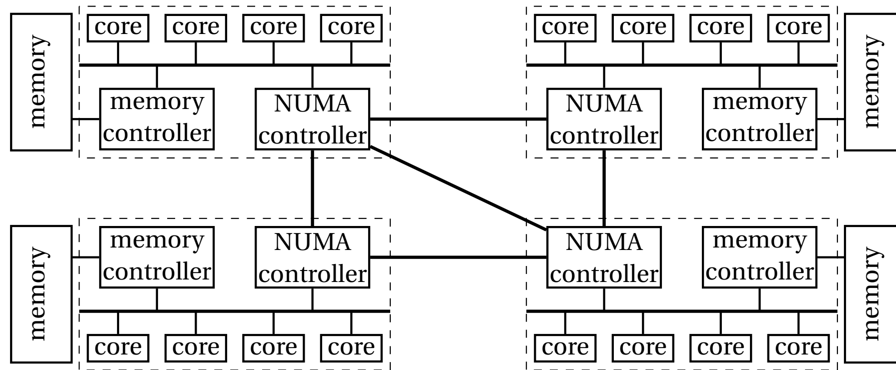

Figure 2.3: The architectural view of a NUMA architecture

coherency makes programmation easier but is expensive, an efficient system should make every inter-core communication explicit and avoid to rely on hardware coherency.

As a consequence, we can assume that (i) to bring the best performance, nontheless the software must avoid inter core sharing, but also the hardware must do. Especially, we describe in Section 2.2 that with a large number of core, hardware components, like memory, must be partitioned in private subcomponents instead of shared.

Also, we can assume that (ii) to bring the best performance, all the sofware layers must be aware of the communication mechanisms their actions imply and use them explicitely. We describe in Section 2.3 that the virtualization layer makes it difficult.

2.2 Non Uniform Memory Access architectures

As we show in Section 2.1, cores synchronizations cause the serialization of operations and hence, poor performance. We show that a good software design reduces the number of core synchronizations on shared data, which improve application performance. However, in a simple multicore architecture, as illustrated in Figure 2.1, the cores still share hardware components. Indeed, each core accesses the memory through a single memory controller and a single system bus. Thus, even if each core accesses to its own private data space, all the cores still share the same memory controller. Since this controller serves the core requests sequentially, it causes the serialization of operations.

The local memory caches of the cores serve a large part of the memory requests. When there is no more than about ten cores in the machine, the number of memory requests sent to the memory controller is then too low to cause a serialization. By contrast, in a machine with several tens of cores, the cache mechanism is not sufficient anymore. It then becomes necessary that several cores can access the memory concurrently. In a *Non Uniform Memory Access* (NUMA) architecture, the available cores and memory banks are partitioned in several nodes, with each node having a separated bus and dedicated memory controllers. All the nodes are bound together with an internal network called the *interconnect*. This ar-

chitecture is illustrated in Figure 2.3. Using several nodes and thus several buses spreads the memory load across the nodes.

This section first presents how the cores communicate in a NUMA architecture. In particular, we describe the HT Assist cache coherency protocol: an extension of the MOESI protocol for NUMA architectures. We also describe the NUMA hardware that we use for our evaluations. We then explain how Linux handle the distributed aspect of the memory. To this end, we present two NUMA memory management policies provided by the Linux kernel. These explanations help to understand what aspects of a NUMA hardware impact the application performance. Finally, we present some existing works that focus on how to use the NUMA architecture specificities to bring good applications performance.

Architecture details

In a NUMA architecture, there are several memory controllers that can serve requests concurrently. Each of these controller serves the requests to its associated memory banks. A *cache coherent NUMA* (ccNUMA) architecture presents these memory banks as a single address space to the software: the first gigabytes stand for the memory banks of the first node, the following gigabytes for the next node and so on. Thanks to this unified address space, legacy applications can execute on ccNUMA computers. In the remainder of this thesis, we focus exclusively on ccNUMA architecture. We thus simply use the term NUMA instead of ccNUMA.

As suggested by the architecture name, the memory access time, in a NUMA architecture, varies depending on the memory bank a core access to. Intuitively, we can assume that a core access a memory bank of this node faster than a memory bank of a remote node. In Chapter 4, we explain that several other factors may impact the memory access time. As a result, accessing to the unified memory address space without considering the underlying memory banks located in different nodes may lead to poor application performance.

In this part, we describe how a NUMA hardware exposes a unified address space out of several memory banks accessed through independent memory controllers. We also give some key characteristics of the hardware we use for our evaluations.

HT Assist protocol

As we explain in Section 2.1, in a modern multicore, the cores access to the memory through a cache hierarchy. A cache coherency protocol maintains sequential consistency for the memory accesses. A usual way to achieve this consistency is to implement a reader-writer lock for each cache line. This lock ensures that at any given time, either one cache contains the cache line and can modify it or many caches contain a read-only copy of the cache line. When the core requires a cache line, the cache controller acquires the lock of this cache line in the appropriate mode.

To acquire a lock in a simple multicore architecture, the cache controller broadcasts messages, called probes, to the other cache controllers. This approach works well with a small

number of cores. In a NUMA architecture with tens of cores, the broadcast approach would saturate the communication links. Thus, using the MOESI cache coherency protocol would be inefficient.

The AMD implementations of the NUMA architecture [23] do not apply the MOESI protocol to all the cores of a machine. Instead, only the cores of the same NUMA node synchronize between each others with the MOESI protocol. Since the number of cores in a single NUMA node is low, using the legacy MOESI broadcast inside a node does not saturate communication links.

Using the MOESI protocol inside each NUMA node is however not sufficient to ensure a sequential consistency over the whole machine. Specifically, the nodes still have to synchronize in two cases:

Read case If a core reads a cache line, the cache controller has to acquire the lock in read-only mode. If no other core in the local node owns the lock, then another core in a remote node may own the lock in the read-write mode.

Write case If a core writes a cache line, the cache controller has to acquire the lock in read-write mode. If no other core in the local node already owns the lock in read-write mode, then other cores in a remote node may own the lock.

For these two cases, the nodes must synchronize to ensure they respect the reader-writer lock invariants.

The NUMA node synchronization has the form of another reader-writer lock, this time at the node granularity. The invariant of this lock is that at any time, either several NUMA nodes can own the lock in read-only mode or at most one NUMA node can own the lock in read-write mode. The synchronization for the **Read case** thus translates into a read-only lock acquisition. Similarly, the synchronization for the **Write case** translates into a read-write lock acquisition. We can notice that the node grained lock and the core grained locks are tied together by two logical relations. If a core owns the core grained lock in read-write mode, then the node of this core owns the node grained lock in read-write mode. Also, if a core owns the core grained lock in read-only mode, then the node of this core owns the node grained lock.

The AMD implementation of the reader-writer lock between the NUMA nodes is called HT Assist. In this implementation, each cache line has an associated NUMA node called the *home node*. The home node stores, for each of its associated cache line, a corresponding state. There are 4 possible states. These states have the same name that the MOESI protocol states, however, they have slightly different meanings. The state Exclusive/Modified represents a lock in read-wite mode. The states Owned and Shared represent a lock in read-only mode. The state Invalid represents a lock not acquired[2].

[2] The AMD documentation [23] actually defines a fifth state called Shared-1. However, it seems that this state is not used, whether in the protocol description or in the experimental evaluation.

	No owner	One owner	Many owners
Clean copy	Invalid		Shared
Dirty copy		Exclusive/Modified	Owned

Table 2.3: HT Assist States

	Invalid	Shared	Owned	Exclusive/Modified
Read-only	-	-	probe owner	probe owner
Read-write	-	broadcast	broadcast	probe owner
Eviction	-	broadcast	broadcast	probe owner

Table 2.4: HT Assist probing policy of the home node

The HT Assist protocol thus implements a reader-writer lock between the nodes for each cache line. Additionally, the HT Assist protocol tries to minimize the number of messages that the NUMA nodes exchange. From this perspective, the reader-writer lock state indicates if no, one or many nodes have a copy of the cache line. Furthermore, the HT Assist states also indicate if the cache line is clean or dirty, and thus if it can be fetched from main memory. This indication matters because the home node, which stores the state of its associated cache lines, is also the node where the cache line has its original main memory copy. Fetching a cache line from main memory then means that the home node does not have to probe any other NUMA nodes and can respond to the acquiring node. Table 2.3 shows, for each state, how many nodes can have a copy of the cache line and if the copy is clean or dirty. For the states storing a dirty copy of the cache line, the home node also stores the id of the NUMA node to probe to fetch the data, called the *owner node*.

To acquire the lock of a cache line, a NUMA node first computes the id of its home node, basing on the cache line address. The acquiring node then probes the home node indicating if it requires a read-only or a read-write lock. Based on the required lock mode and on the cache line state, the home node decides what probe messages it sends.

If the acquiring node requires a read-only lock and the cache line is clean, so Invalid or Shared state, the home node serves the cache line from its main memory and updates the cache line state to Shared. If the cache line is dirty, so Owned or Exclusive/Modified, the home node probes the *owner node* and updates the cache line state to Owned. The owner node sends its cache line copy to the acquiring node. Additionally, if the node was in the Exclusive/Modified state, it ensures that its cores update their MOESI state from Exclusive to Shared and from Modified to Owned.

If the acquiring node requires a read-write lock and many NUMA nodes may own a copy of the cache line, so Shared or Owned states, the home node broadcasts an invalidation probe for this cache line. If the cache line is in the Exclusive/Modified state, the home node only sends an invalidation probe to the owner node. In both cases, the home node updates the cache line state to Exclusive/Modified and sets the acquiring node as the owner node. Table 2.4 summarizes the probes that the home node sends for each combination of

required modes and cache line states.

Because the home node of a cache line always knows its associated state, it sends probes only if the cache line cannot be served from main memory. Additionally, the home node only sends directed probes when possible, as opposed to broadcast probe. However, the home node can only store a limited number of cache line states. These states are stored along with the owner node of a cache line in the last level cache of the node. This storage area is called the *cache directory*.

When a cache line is loaded from the main memory for the first time, the home node creates a new corresponding entry in the cache directory. If the cache directory is already full, an older entry is evicted. The HT Assist protocol however requires that any cache line that has a copy in any cache also has an entry in the cache directory. A cache entry eviction thus implies that the corresponding cache line is invalidated from all the caches.

The home node uses the state of the evicted cache line to decide to which node it sends invalidation messages. If the cache line may be owned by many NUMA nodes, so Shared or Owned states, the home node broadcasts invalidation probes. Instead, if the cache line is in the Exclusive/Modified state, the home node only sends an invalidation probe to the owner node. Table 2.4 summarizes the probes that the home node sends on a cache line eviction depending on the cache line state.

Using both the node grained lock for the whole NUMA machine and a core grained lock in each NUMA node guarantees that at most one core in the machine modifies a cache line at a time. Using these locks also guarantees that the cores always read the freshest state of a cache line, which is a way to provide sequential consistency. By this mean, the HT Assist protocol ensures the memory accesses from all the cores are sequentially consistent, only using a low number of internode messages.

Synthesis

In Section 2.1, we show that understanding the MOESI protocol helps to design software that use the hardware more efficiently. In the same way, a software understanding the HT Assist protocol is more likely to avoid performance killer patterns:

- Transferring a cache line between two NUMA nodes. This happens if two cores of different nodes modify the same cache line. The performance loss is even larger than in MOESI cache line sharing since a HT Assist read-write acquisition may involve up to three NUMA nodes. Also the internode messages go through the interconnect, which is slower and has less bandwidth than intranode links.

- Misplacing data in the main memory. Surprisingly, the HT Assist cache coherency protocol makes the main memory data placement quite important for performance. Indeed, when a NUMA node fetches a cache line, it reads it from main memory, even if another node owns a copy of this cache line. However, the cache line copies have to be clean. More importantly, an internode communication always involves the home

Cache		Memory		
			1 thread	48 threads
L1 cache	5 cycles	Local	156 cycles	697 cycles
L2 cache	16 cycles	Remote (1 hop)	276 cycles	740 cycles
L3 cache	48 cycles	Remote (2 hops)	383 cycles	863 cycles

Table 2.5: Cache and memory access latency on the experimental machine.

node of the cache line. Furthermore, the home node of a cache line depends on the cache line memory address.

By knowing the NUMA aspect of the hardware, the software can thus take efficient placement decisions. By placing the tasks that work on the same data on the same NUMA node, the software avoids internode cache line transfers. By placing its cache lines on the same node than the tasks using them, the software decreases the memory access latency. Also, spreading its cache lines over several NUMA nodes, the software ensures that the memory pressure spreads evenly and that no home node saturates.

Hardware used for the evaluation

This section presents the NUMA machine used in the evaluation. We use an AMD machine having 8 NUMA nodes with 6 CPUs/16 GiB per node for a total of 48 cores and 128 GiB of RAM. In summary, this machine has four Opteron 6174 sockets, each one containing two NUMA nodes. Each node is connected to 16 GiB of RAM through a memory controller having a maximum throughput of 13 GiB/s. The 6 CPUs of a node share a unified L3 cache of 5 MiB. A CPU of a NUMA node runs at 2.2 GHz. It has two L1 caches of 64 KiB each (a data and a code cache), and a single unified L2 cache of 512 KiB.

The nodes are interconnected by HyperTransport links, with a maximum distance of two hops. The bandwidth between nodes is asymmetric, with a maximum bandwidth of 6 GiB/s. Two of the nodes (nodes 0 and 6) are connected to a PCI bus. The network and the disk that contains the system files are connected to the bus of node 0. The disk that contains all the benchmarks and the datasets is connected to the bus of node 6.

Table 2.5 reports the time to access the caches and the memory. In order to measure the memory access latency, we present two results. With 1 thread, we measure an uncontended case, in which a single thread accesses a single NUMA node. With 48 threads, we measure a contended case, in which 48 threads access the same NUMA node. This result shows that the diameter has only a small impact on performance (276 cycles for a 1-hop access versus 383 cycles for a 2-hops access in the uncontended case). On the contrary, we observe that a contended memory controller drastically slows down memory access latency (697 cycles to access a local NUMA node when this last is contended).

On the machine we use, accessing a saturated node, even the local one, is twice more expensive than accessing a remote but uncontended node. To reach good performance, it is

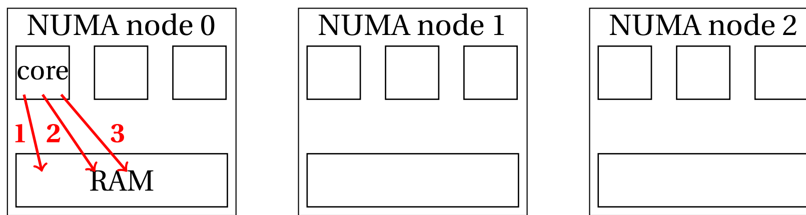

Figure 2.4: First touch allocation policy. Each red line is an allocation. Here the core allocates memory in its local node.

thus more important to avoid node saturation than maximizing locality.

Linux NUMA policies

This part presents some default NUMA memory management policies. We call a NUMA memory management policy, or simply NUMA policy, a strategy that software uses to place its data on the NUMA nodes. The software can be the end user application or another layer software, often the operating system. An end user application uses a NUMA policy suited for its own memory access pattern. In this thesis, we focus on NUMA management policies working for a vast set of applications. Thus we describe here two NUMA policies that the Linux kernel provides.

First-Touch

The default policy is the First Touch. It aims at maximizing the data locality, which avoids expensive remote accesses and prevents interconnect saturation. The idea behind this policy is that the core that access a data item the first is probably the exclusive accessor of this data. This is typically the case for the variables of the stack. As Linux lazily allocates application memory by only reserving unmapped virtual addresses, a first access to data leads to a page fault. When the Linux kernel handles the page fault to make the actual memory allocation, it knows which core is making the access. It thus can allocate space in the local NUMA node. An allocation in another node is also possible if there is not enough free memory in the local node. This allocation policy is illustrated in Figure 2.4.

This allocation strategy actually maximizes locality for many applications. Indeed, in many applications, the threads use mostly the memory areas that they allocate and rarely the memory areas that the other threads allocate.

However, some applications use a master-slave design, in which the master thread allocates and initializes a large portion of the memory. The slave threads then perform computations on this memory area. In this scenario, all the memory is allocated on a single NUMA node. When the slave threads start to access this memory, they cause an important saturation of the memory controller, leading to terrible performance.

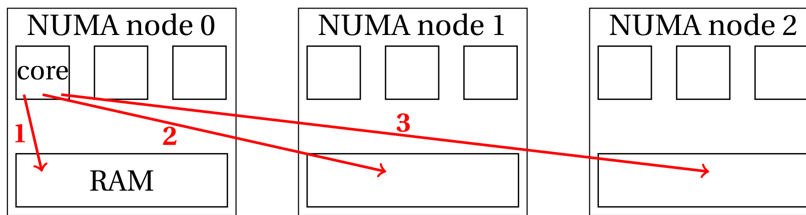

Figure 2.5: Round-4k allocation policy. Each red line is an allocation. Here the core allocates memory in every node one after another.

Round-4k

The Linux kernel offers the possibility to spread the memory allocation across a set of NUMA nodes regardless of the first accessor. This is called the round-4K policy and has to be enabled by the user for each application using this policy. For these applications, the memory is allocated in a round robin fashion over a set of given NUMA nodes. When the memory cannot be allocated on this set of nodes, it is allocated from the other nodes. This policy ensures that the allocated space is evenly spread over the NUMA nodes. Thus, if each core accesses randomly to the allocated memory, or a large enough subset of memory, then the memory load is also evenly spread over the NUMA nodes. As a result, no NUMA node saturates. The counterpart is that each core only makes a small number of local accesses. Figure 2.5 illustrates this allocation policy.

Synthesis

Linux provides two NUMA policies, first-touch and round-4K. The first one assumes that threads mostly use the memory they allocate. If so, the first-touch maximizes the amount of local access and prevents node saturation. The round-4K policy only assumes that threads access randomly to a large enough subset of the memory they allocate. If so, this policy prevents saturation. However, the round-4K policy does not maximize the amount of access locality.

Each of the two presented policies is better suited for one type of application, but none of them is absolutely better than the other one. Because of that, Linux lets the application administrator decide which policy to use with a dedicated tool: `numactl`. Additionally, the Linux kernel provides an API to let userland software finely tune on which node to allocate memory or execute a thread. This interface allows either the application programmer or the system administrator to choose how to deal with the NUMA architecture specificities.

Related challenges

In the previous section, we describe the NUMA architecture. Specifically, we show that the placement of both the tasks and the data is a critical factor for the application performance. In this section, we present works that aim at leveraging high performance from a NUMA hardware. As we explain in Section 2.2, we focus on NUMA policies that work for a vast set

Solution	Hardware		Software
	Memory locality	Memory contention	
Lock Cohorting	+	-	
Everyth. about Synchro.	++	-	lock contention
NumaGiC	++	+	object allocation
Carrefour	+	++	memory access

Table 2.6: Works related to NUMA architectures

of applications. We thus do not present any work where a NUMA policy is especially crafted for a particular application.

Table 2.6 enumerates the works we present. We choose to present and classify these works from the memory placement perspective, instead of the task placement perspective. As we explain is Section 2.2, an efficient NUMA memory placement is a compromise between a high access locality and a low contention on the home nodes. We show the importance these works give to each of these two memory placement aspects. Knowing the details of a NUMA hardware is, however, not sufficient to achieve good performance. A NUMA policy also has to be well suited for the software that makes the memory accesses. We thus report, as a third classification criteria, what aspect of the software the presented works take into account.

Lock Cohorting

In a multicore architecture, whether NUMA or not, communication between cores has a major impact on performance. One of the most common communication patterns is the lock acquisition and releasing. For this reason, many works [45, 2, 24, 59, 1] address the problem of efficient locking. More precisely, these works address the scalability of a locking system. This scalability is often related to the amount of communication a lock acquisition generates. While this aspect of locks has a great impact on performance, it is not specific to the NUMA architecture.

In addition to the scalability of a locking system, Dice et al. also tackle, with the Lock Cohorting technique [28], the problem of task scheduling. Indeed, a locking system determines in what order the threads execute by choosing the lock acquisition order. Usually, a lock protects a shared data item that several threads access concurrently. Additionally, some of the accessing threads modify the shared data item, otherwise no lock would be needed. The lock acquisition order thus also determines the shared data access and modification order.

In a NUMA architecture, sharing and modifying a data concurrently inside of a NUMA node is far less expensive than doing so between several NUMA nodes. In a set of threads that want to modify the same data item, it is thus more efficient to execute the threads in a same NUMA node before to execute the threads of the other nodes. By doing so, a locking system minimizes the amount of internode cache line transfer.

The Lock Cohorting is a technique that combines two well-known types of lock into a cohort lock. When released by a thread, a cohort lock gives ownership to the threads of the local NUMA node in priority. To ensure fairness, a cohort lock transfers ownership to a thread of another node after a configured number of releases. A cohort lock is composed of a global lock G and one local lock S_i per NUMA node. To acquire the cohort lock, a thread must acquire both S and G, in this order. To release the cohort lock, a thread first checks if some other thread is waiting for S. This other thread is thus a local concurrent. If so, it releases S and transfers the G ownership to this local concurrent. Otherwise, it releases both S and G.

The Lock Cohorting technique increases the cache locality of an application. It does so by scheduling threads on the same NUMA node together when these threads compete for the same lock. The cache locality slightly differs from memory locality as the main memory is not necessarily implied. However, as we see in Section 2.2, even a cache to cache transfer always implies the home node of the cache line. This home node can saturate when it has to handle too many HT Assist requests. The Lock Cohorting technique does not consider this aspect of NUMA architecture. Moreover, a cohort lock does not use additional properties of the end user application to improve efficiency.

Everything about Synchronization

The Lock Cohorting technique is not the only work that tackles the locking efficiency in a NUMA architecture. Other designs [27, 55] has been proposed to efficiently lock in this architecture. David et al. do not propose a new lock design but instead, give a quite complete study of the relations between the locking system, the hardware and the software. This study, named Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask [26], evaluates and describes the behavior of several state-of-the-art locks over several types of hardware, and especially NUMA hardware, with several workloads. As this work does not give any new solution but focuses on understanding how existing solutions behave, we here analyze the observations the study makes.

The first observation is that, as expected, synchronizing cores of different nodes kills the performance. More formally, the acquisition of a lock owned by a thread on a remote node can be a dozen times more expensive than if the owner thread was on the local node. Interestingly, this phenomenon worsens as the lock gets (i) more contended and (ii) more complex. This second assertion does not stand if the complexity is added to make the lock NUMA aware.

The second observation is that, considering the NUMA aspect of the hardware at a high level is not always sufficient. The study highlights a surprising consequence of the HT Assist protocol described in Section 2.2. Even if a synchronization, or any concurrent memory modification, is limited to the cores of a single node, it may lead to internode broadcasts. Indeed, if a cache line is in the Shared or Owned state, a modification of this line forces the home node to broadcast invalidation probes. This occurs even if only the local cores own a copy of the cache line. David et al. propose a fix that consists in forcing the cache line state to Exclusive/Modified with a dedicated instruction.

The third observation is that the lock complexity often adds overhead in NUMA architecture, but can improve performance under specific conditions. As David et al. state in the first observation, using a lock across NUMA nodes is more expensive when this lock is more complex. However, under high contention, these locks perform better than the simple locks.

This study also gives other observations that are not related to the NUMA architecture. We choose to elude them and summarize the three observations that we present. First, software aware of the high level NUMA topology is more efficient by avoiding internode data sharing, increasing the locality. Second, software aware of the low level NUMA details is more efficient by avoiding hardware mispredictions or degenerated behaviors. Third, software aware of the end user application details, like the locks contention, is more efficient by choosing an appropriate strategy. This third point is not specific to NUMA. However, it is closely related to the necessity for a system to know the needs of the end user application to choose the appropriate strategy, whether a lock implementation or a NUMA policy.

NumaGiC

The locking system is not the only software layer that can take advantage of the NUMA topology to improve application performance. A managed runtime environment, typically a Java Virtual Machine (JVM), usually handles the low level memory allocation details. This is especially the case when the runtime provides a garbage collection system. In a runtime with a garbage collector, some runtime threads walk the objects graph of the application to separate reachable objects from unreachable objects. These threads then copy the reachable objects in a new memory zone, then free the old zone. The runtime threads, called GC threads, execute either concurrently with the application threads, called mutator threads, or during a stop-the-world phase. When the application objects graph is large, walking the whole graph becomes expensive and affects the overall application performance. Modern JVMs use several threads that walk the objects graph concurrently.

When the application objects graph is spread over several NUMA nodes, a GC thread can blindly follow internode references. Following such a reference implies that the core fetches data from the memory of a remote node through the interconnect. If too many GC threads use the interconnect, this last saturates, causing an important performance drop. Gidra et al. address this issue with NumaGiC [34], a garbage collector designed to improve the memory locality during a collection.

NumaGiC improves the collection memory locality by using four heuristics. The first heuristic is to allocate new objects on the same NUMA node than the allocating mutator thread. The reason is similar to the one behind the first-touch NUMA policy: a thread often works on the objects it allocates. This implies that a mutator likely allocates new objects and binds them with references, which happen to be local references. As a second heuristic, the GC threads process local objects in priority. The NumaGiC is designed so a GC thread knows the home node of an object basing on its virtual address. If the object has a remote home node, the GC thread sends its reference to the GC threads of this remote node. Only if a GC

thread becomes idle, it starts to steal work to remote GC threads. These first two heuristics improve the memory locality by making the GC threads process mostly local references.

The third heuristic is to copy reachable objects to the node of the GC thread that collected it. If the GC thread processes local references, this heuristic preserves the memory locality. If the GC thread steals work, this heuristic rebalances the memory load. Indeed, if too many objects are located on the same node, mutator threads likely access this unique node, possibly creating memory saturation. In this case, the GC threads of this overloaded node have more work than the GC threads of the other nodes. These last threads thus steal work and then rebalance the memory load.

The fourth heuristic is to copy old reachable objects to the same node they were on before the collection. In NumaGiC, an object is old if it survives to 3 collections. This fourth heuristic moderates the third one and prevents GC threads to copy old objects back and forth across the NUMA nodes when there is only a negligible imbalance.

The NumaGiC thus focuses on memory access locality by (i) ensuring objects that reference each others are on the same node, (ii) ensuring that GC threads mostly process local references and (iii) preventing useless object migrations. Additionally, NumaGiC prevents a memory imbalance scenario by rebalancing young objects location during the garbage collection. While the last three heuristics concern the GC threads, the first heuristic takes into account and modifies the objects allocation made by the mutator threads.

Carrefour

The three previous works improve the memory access locality and only place a few weight on memory access balance. For the synchronization study [26], this makes sense because the target is the lock itself. It seems unlikely for a lock to generate so many memory accesses that it saturates a home node. The NumaGiC is also a special case as the GC threads explicitly avoid accessing to the memory of a remote node. Furthermore, only the Lock Cohorting technique aims to improve the end user application memory accesses. NumaGiC also improves the end user application memory locality and balance but only as a side effect.

Dashti et al. present Carrefour [25], a NUMA policy. The only goal of the Carrefour policy is to improve the end user application memory accesses. Also, Carrefour puts emphasis on balancing the memory accesses over improving access locality. We actually use this work as a base policy later in this thesis. For this reason, we describe here precisely how it works. We present an experimental evaluation of this policy in Chapter 4.

Carrefour is a dynamic memory placement policy for NUMA architectures. This contrasts with the first-touch and round-4K policies that we say static: when a memory area is allocated, the first-touch or round-4K policy decides on what node to allocate. Instead, Carrefour dynamically migrates memory pages, which have already been allocated, in order to improve the memory access balance, while improving the memory access locality as a secondary goal.

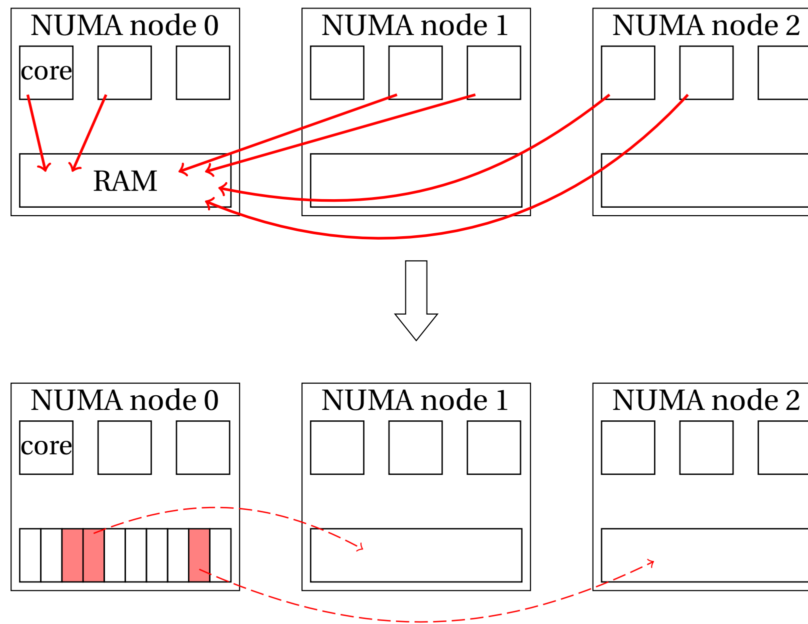

Figure 2.6: Carrefour interleaving heuristic: when a memory controller is overloaded (top part of the diagram), the hot pages of this memory controller (the read rectangles in the bottom part of the diagram), are spread between the NUMA nodes

To this end, Carrefour monitors the memory access pattern of the application threads using hardware counters. A first set of counters, called Performance Monitoring Counters, accounts the number of memory access per second. These counters are cheap to use and Carrefour uses them constantly. If the number of memory access per second exceeds a fixed threshold, then Carrefour enables a second set of hardware counters: the Sampling Counters. The Sampling Counters are more expensive to use, so Carrefour enables them only if the application uses the memory intensely. Periodically, the Sampling Counters take the current instruction of the monitored application and report to Carrefour what memory address it accesses. With this information, Carrefour builds a map between the threads of the application and the memory pages they access to. Since Carrefour builds this map with sampling, it is only statistically valid.

Given this access map, Carrefour implements three heuristics to migrate or replicate the hottest pages i.e. the most accessed pages. The first heuristic implemented by Carrefour is the *interleaved heuristic*. Carrefour executes this heuristic when it detects that memory controllers are overloaded. It randomly migrates hot pages from overloaded nodes to underloaded nodes. By doing so, this first heuristic balances the memory access load between the NUMA nodes. Figure 2.6 illustrates this first heuristic.

Carrefour also implements two heuristics that it triggers when it detects that the interconnect saturates. Indeed, balancing the memory accesses between the nodes does not reduce the load on the interconnect. Quite the contrary it increases it. The *migration heuristic* mi-

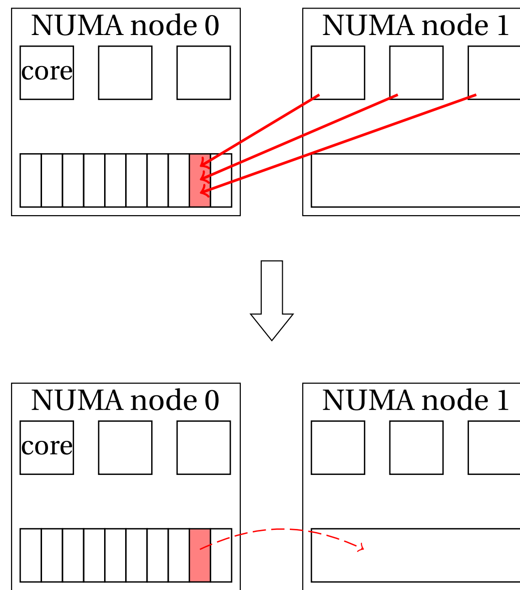

Figure 2.7: Carrefour migration heuristic: when a hot page is accesses only by remote cores of the same node (top part of the diagram), this hot page is migrated on the accessing NUMA node

grates hot pages that are remotely accessed by only a single node towards the node that performs the accesses. The *replication heuristic* replicates hot pages that are accessed in read-only mode by a set of threads. These two heuristics improve the memory access locality. As a result, Carrefour reduces the memory access latency and reduces the load on the interconnect. Figure 2.7 illustrates the migration heuristic.

Carrefour is thus a compromise between the first-touch and round-4K policies. On one side, it balances the memory load across the NUMA nodes, like the round-4K policy. Unless the round-4K policy, Carrefour only interleaves the hot pages. On the other side, it improves the memory access locality, like the first-touch policy. Carrefour does so by migrating the pages accessed by a unique node towards this node. However, Carrefour is a dynamic policy. Carrefour first observes the application memory access pattern, then modifies its memory layout. The advantage is that Carrefour adapts to an application that changes its behavior over time. The downside is that Carrefour is unable to take immediate decisions. It must work aside from another static policy.

Conclusion

The presented works address the problem of tasks and memory placement in a NUMA architecture. This placement problem is difficult because an efficient placement depends on several factors. The memory access locality and the memory access balance are high-level concerns and largely impact the end user application performance. To achieve a good access locality and balance, the software has to (i) know the hardware details as precisely as possible and (ii) know the application behavior as finely as possible.

The condition (i) is necessary for the software to find a good compromise between memory locality and memory balance. Placing a memory page on the local node instead of sending it on an unbalanced node may or may not be a good decision depending on the bandwidth of the interconnect and the memory controller. Additionally, by knowing the details of the hardware, the software stack avoids degenerated behavior of the hardware.

The condition (ii) is necessary for the software to use the best suited NUMA policy. As we explain in Section 2.2 and as we confirm it in Chapter 4, there is no policy that outperforms all the other policies for all applications. Knowing the behavior of the application is then essential to bring good application performance. The software can know the application behavior either with monitoring techniques, like Carrefour, or with administrator clues, like `numactl`.

We show in Section 2.3 that system virtualization prevents the virtualized software to know the hardware details. Additionally, the software responsible of the virtualization only sees virtual machines as a black box and thus cannot know the end user application behavior.

2.3 System virtualization

Aside from the evolution of hardware allowing the execution of more complex applications, the way to use computers also change. An increasing number of users now prefers to execute their applications on distant and allocated on demand hardware, called the Cloud. We call the Cloud, the aggregation of machines owned by a company called *the provider* which are rented to users called *the clients*.

A provider can allow its clients to use the machines in various form. We focus on the Infrastructure As A Service form, where each client rents a complete computer. With this type of service, the client can install its own operating system on the computer and has full privileges on it.

Usually, the provider does not give directly a physical machine to its clients. The first reason is that the Cloud machines are powerful server while the client generally only needs a fraction of this computing power. The second reason is that giving the full privileges on the provider servers is a security issue.

Instead of directly giving access to the physical machine, the Cloud provider executes one or several computer simulations on its servers. As the machine simulation, called *virtual machine* behaves exactly like a regular computer, the provider can rent it to the client. By executing several virtual machines on a single physical machine, the Cloud provider splits the computing power of a server between several clients. Also, giving full privileges to a virtual machine is not a security concern since the client cannot escape the machine simulation.

Executing virtual machines is a technique called *system virtualization.* In this section, we explain how the system virtualization works and what challenges it brings. In the first part, we describe, at a coarse grain, the different methods ones can use to achieve system virtu-

alization. Then, we focus on one of these methods, called the *Hardware Assisted Virtualization*, and explain specific details we then use in the remainder of the document. Finally, we present some challenges brought by the virtualization and how existing works address them.

Technical details

The system virtualization is neither a recent technology nor a technique specific to the Cloud computing. Quite the contrary, virtualizing entire computers has been used in the first place to execute software designed for an Instruction Set Architecture (ISA) different that available hardware. The ability to execute software compatible to ISA *A* on hardware with an ISA *B* reduces the costs of porting software on multiple architectures. The ported software is called the *guest* and the computer executing the virtual machine is called the *host*.

In second place, system virtualization has been largely used for operating system development and debugging. In this case, the virtual machine may, or not, have the same ISA than the physical machine. The main goal is to simulate a machine with the same behavior than a physical machine while providing useful debugging mechanisms, like the ability to breakpoint or to inspect the processor state.

With the operating systems becoming more complex, involving more operations, their programmers needed to execute virtual machines more quickly to avoid slowing down the development process. The virtualization techniques thus evolved to become more efficient.

The system virtualization became so efficient that it is now possible to execute usual software inside a virtual machine with acceptable overhead. It is then suitable for Cloud computing usage. In the context of the Cloud computing, the execution speed of virtual machines directly impact the benefits of the provider: the more its virtual machines are effective, the more it can execute virtual machines on the same hardware for the same quality of service. The system virtualization techniques thus evolved again to become even more efficient, this time with the support of processors manufacturers.

This thesis focus on the performance of virtualized applications. In this first part, we then describe successive virtualization techniques from the most naive to the most effective from the angle of performance.

Cycle accurate simulation

The most obvious solution to provide a virtual machine is to use a program that simulates the behavior of each chip and each wire of the actual machine. This kind of program is called a *cycle accurate simulator*. A cycle accurate simulator can thus simulate hardware with any desired ISA. Additionally, the behavior of the virtual machine matches exactly the behavior of a physical one.

Computers are however complex beasts, and to execute one instruction, the processor activates a plethora of chips and wires. Simulating each of them requires an important com-

puting power. Despite the accuracy of the virtual machine behavior they execute, cycle accurate simulators are thus extremely slow.

Hardware emulation

As stated at the beginning of this section, before the Cloud computing, most of the virtual machines users only needed to port software to multiple ISA or to develop operating systems. For these tasks, having an exact reproduction of the physical machine circuitry is not necessary. Having a virtual machine functionally equivalent to the physical one is actually sufficient.

A simple technique to avoid simulating the whole hardware is to translate the guest instructions into what should be executed instead on the host. The program in charge of virtualization, called *emulator*, reads each instruction of the virtualized software, whether operating system or userland program. When reading an instruction, the emulator decodes it and executes a code with a corresponding behavior. The executed code is a part of the emulator and hence, compatible with the physical machine ISA. Additionally, any privileged instruction of the guest is replaced by an emulated behavior.

An example of emulated behavior is the disk controller emulation. Typically, the processor usually communicates with disk controllers by sending word sized commands on the I/O bus. To program a disk read, the processor sends a command indicating where to read on the disk, then another command indicating where to write on the memory, then a command to trigger the read. On the x86 architecture, sending a command to a device uses a privileged instruction. To emulate a disk controller, the emulator maintains the state of the virtual disk controller in a structure in memory. It then decodes each of the privileged I/O instructions and updates the state of the virtual controller accordingly. The final I/O command indicates the emulator to look at this virtual state to fetch the disk sector and memory offset to use. The emulator finally reads the virtual disk, which can be a file or a dedicated physical disk of the physical machine.

An emulator thus maintains a set of virtual devices. It updates them in response to privileged instructions issued by the virtualized software. These virtual devices are not simulated, like in a cycle accurate simulator. Instead, they only behave like an actual device would do. For this reason, hardware emulation is more efficient than cycle accurate simulation. Additionally, it is still well suited for operating system development since the state of the virtual device can be inspected at any moment during the execution. Also, inserting breakpoints is an easy task: it only requires to insert hooks between two virtual instructions decoding.

Dynamic Binary Translation

Hardware emulation is more efficient than cycle accurate simulation but still can be improved. Indeed, decoding each virtual instruction induces a significant overhead. In particular, an emulator decodes the instructions in the body of a loop at every iteration. It is very unlikely the guest modifies its own code from an iteration to the other.

Caching this instruction decoding for later use is a powerful optimization. A common way to cache these decodings is named the *Dynamic Binary Translation*. A DBT emulator decodes complete blocks of instructions of the guest and translate them to blocks of instructions compatible with the host ISA. Most ISAs often provide very similar functionalities so most of the guest instructions can be translated into one or a few host instructions. Guest instructions that cannot be easily translated or privileged instructions are replaced by calls to the emulator code. The resulting block of instructions is then executed and stored in memory for future use. If the guest wants to execute this block of instructions later, it is already translated and can be executed directly.

Since a decoded block is directly executed by the host processor, unprivileged guest code induces no overhead once translated. Usage of privileged devices however, still requires additional operations for the host to update the virtual devices state. Translating the guest blocks of instructions when encountered for the first time also adds overhead. Additionally, the counterpart of the improved performance is that inspecting the hardware state is not possible anymore for the CPU. Indeed, the CPU is not emulated by a DBT emulator since the host CPU is directly used. Devices can however still be inspected and breakpoints are still possible by inserting hooks in the produced instruction blocks.

Dynamic Binary Translation for same ISA

A special case of DBT occurs when the guest and the host have the same ISA. This is typically the case for operating system development or debugging. In this case translating an instruction block is straightforward. The DBT emulator only replaces privileged instructions by calls to the emulator code. The processor hence executes most of the guest code unmodified. In this common scenario, the DBT emulation only causes a small performance overhead.

In the remainder of this document, we call *hypervisor* or *Virtual Machine Monitor* (VMM), an emulator specifically designed to execute virtual machines with the same ISA than the physical machine [60, 36, 35, 17]. We now focus exclusively on hypervisors since Cloud computing virtualization relies on these programs.

Paravirtualization

While a hypervisor, thanks to hardware emulation or DBT, executes unprivileged instructions with a reduced overhead, emulating devices remains a slow process. Emulating a disk, for instance, requires to emulate the complete behavior or a disk controller. As stated earlier, a processor requires a disk read by sending several word sized commands on the I/O bus with privileged instructions. Each command updates the state of the disk controller and the last command tells the controller to actually perform the read. To emulate this device correctly, the hypervisor needs to catch every I/O instructions to update a virtual disk controller. This causes an important overhead.

To remove the device emulation overhead, some hypervisors trade their ability to mimic the exact behavior of a physical machine against the ability to cooperate with the guest op-

erating system. This technique is called *paravirtualization* [8, 7, 53, 31, 42]. A paravirtualizing hypervisor notifies the guest operating system that it executes inside a virtual machine. The operating system, which is modified so it can use paravirtualization, directly requests services to the hypervisor instead of using emulated devices. In the same way we name *syscall*, a request to the operating system, we name *hypercall*, a request to the hypervisor. To read from a disk, a paravirtualization enabled operating system makes a single hypercall that is less expensive that executing several privileged instructions that are caught by the hypervisor. Furthermore, a paravirtualizing hypervisor does not need to maintain virtual device states since there is no more need to delude the guest into thinking that there is an actual device.

By using the paravirtualizaion technique a hypervisor, whether a hardware emulator or a DBT hypervisor, provides access to privileged operations with a low overhead. The counterpart is that the guest operating system has to be modified to replace the low level hardware management routines by hypercalls. It is thus not possible to execute any operating system in a paravirtualized machine. This constraint makes a paravirtualizing hypervisor unsuitable for operating system development, except for paravirtualization enabled operating systems. However, it is adapted to the Cloud computing where performance and security are the main concerns.

Hardware Assisted Virtualization

The paravirtualization technique reduces the number of interceptions the hypervisor makes and thus the number of context switches between the guest and the host. It however has some drawbacks. First, the guest operating system has to be paravirtualization enabled. Second, the paravirtualization works well only for privileged operations that update stateful devices, like disk controllers. This is because the complete emulation of a stateful device requires several interceptions and thus several expensive context switches while paravirtualizating this same device requires only one context switch. For other special but frequent operations, like performing a syscall, the paravirtualization does not help since it results in one context switch anyway.

With the growing amount of Cloud services relying on virtualization, making hypervisors efficient became an economical necessity. Processor manufacturers then began to add virtualization support in their chips [51, 54, 33, 68, 30]. With this hardware assistance, modern hypervisors emulate hardware for unmodified guest systems with good performance. In this thesis, we focus on this last type of virtualization. For this reason, we describe the *Hardware Assisted Virtualization* with more details in the next section.

Architecture details

Modern processors provide assistance for virtualization to hypervisors. This results in the addition of several features to classical processor functionalities. In this section we describe some general concepts related to *Hardware Assisted Virtualization* (HAV). Moreover, this thesis focuses on NUMA architecture, in addition to system virtualization. These archi-

tectures require a specific memory management to be efficient. We thus give more specific details about how the memory management works with HAV. On x86 architectures, the HAV features differ between the two manufacturers Intel and AMD. As we use AMD processors in our evaluation, we focus on this implementation but if the low level aspects of the HAV change, the concepts remain the sames.

Guest mode

The general idea behind HAV is that the processor can operate in two modes: the host mode and the guest mode. These modes are orthogonal to the kernel mode and user mode (protection rings in the x86 nomenclature). In host mode, the behavior of the processor is unchanged. The hypervisor code is executed in host mode where it has full access to the hardware and can execute privileged instructions. The guest operating system and guest applications execute in guest mode. In guest mode, the processor itself emulates the privileged hardware. When the guest operating system executes a privileged instruction, the processor updates a virtual state and answer to the guest without involving the hypervisor. The processor actually implements predefined behaviors for every device. The hypervisor only configures the processor while in host mode to select what behavior the processor should follow when in guest mode. The hypervisor only handles complex operations such as high level virtual machines management.

Technically, an AMD processor boots in host mode. To execute a new virtual machine, the hypervisor configures a memory zone called the *Virtual Machine Control Block* (VMCB). This zone has a predefined layout and the processor can parse it. This layout defines two parts: (i) the control fields and (ii) the guest state. The control fields indicate what behavior the processor should follow when in guest mode: what privileged instruction to intercept, how to emulate interrupts and so on. The guest state contains the values of the processor virtual registers: the registers seen by the guest system. These registers include the general purpose registers, the instruction pointer register but also the x86 control registers and other privileged registers.

After configuring the VMCB, the hypervisor enters in guest mode with the special instruction `VMRUN`. At this point, the host state is saved in a dedicated memory zone while the guest state is loaded from the VMCB. The processor begins to fetch instructions from the address specified by the virtual instruction pointer. If an event occurs that the processor cannot handle by itself, for instance the guest system trying to halt an idle core, it switches back to host mode and resume the hypervisor execution right after the `VMRUN` instruction. This switch back is called a `VMEXIT` in the AMD nomenclature. During the `VMEXIT`, the guest state is saved in the VMCB and the host state is restored. Additionally, the processor indicates in dedicated VMCB fields what caused the `VMEXIT`. Basing on these fields and on the guest state, the hypervisor decides how to handle the event that occurred in the guest.

The `VMEXIT` is a convenient way for the processor to delegate complex management to the hypervisor. A good example of complex management is virtual CPU (vCPU) scheduling. A vCPU is an execution thread that the guest system sees a physical core. Just like an operating system schedules threads on available cores, a hypervisor schedules vCPUs on physical

cores. There are many scheduling strategies and some of them are quite complex. This kind of high level decisions is not achievable by a processor predefined behavior.

In contrast, the processor handles simple operations by itself. An instance of a simple operation is the system call. Since x86 protection rings (user and kernel mode) are orthogonal to host and guest mode, the processor is able to switch from user to guest without implying the hypervisor.

Hardware Assisted Paging

To isolate the guest system from the physical machine and from the other guests, the hypervisor provides memory isolation of virtual machines. This isolation is roughly similar to the memory isolation between processes that an operating system provides. In this part, we quickly remind how memory isolation of processes works in native configurations (without any virtualization). We then describe how memory isolation of virtual machines works in HAV.

In a native configuration, the operating system maintains, for each user process a mapping between virtual and physical addresses. The user processes only see and access to virtual addresses. The operating system ensures that the virtual addresses of several processes never map to the same physical address, which indicates an actual position in memory. By doing so, the operating system ensures memory isolation of processes, since two processes can never access the same physical address.

Technically, when a process accesses to memory, the processor issues a memory request for a virtual address. This request goes through the Memory Management Unit (MMU), which translates the virtual address into a physical address. The request is then forwarded to the memory. The MMU translates virtual addresses by looking into a page table. The page table is a data structure stored in memory and filled by the operating system. On x86 processors, the page table address is indicated by a control register, named `CR3`. The operating system can modify the `CR3` at any moment to indicate another page table, thus changing the used mapping. This is typically the case when context switching from a process to another one.

In a virtualized configuration, there are two levels of memory isolation. In a given virtual machine, the operating system has to isolate the processes from each other. On the physical machine, the hypervisor has to isolate the virtual machines from each other. HAV capable processors achieve that with the *Hardware Assisted Paging* (HAP) feature.

When the HAP is enabled and the processor executes in guest mode, the MMU operates two translations instead of one. First, the processor issues a memory request for a virtual address. The MMU translates this virtual address to a guest physical address: an address on the emulated physical space as seen from the guest system. To operate this first translation, the MMU looks in a page table indicated by the virtual `CR3`. Indeed, just like many other registers, the `CR3` is emulated by the processor when in guest mode. This first page table is filled by the operating system with no intervention of the hypervisor. In a second step, the MMU translates the guest physical address into a host physical address. To operate this second translation, the MMU looks in a second page table indicated by a field of the VMCB

named `nCR3`. The hypervisor is responsible for filling this second page table. The MMU then forwards the request to the memory.

In the remainder of this document, we use the AMD nomenclature to name the different address spaces. We use the term *physical address* to designate the guest physical addresses and *machine address* to designate the host physical addresses.

In the same way an operating system maintains a page table per process and ensures two page tables never map to the same physical addresses, a hypervisor maintains a page table per virtual machine and ensures they never map to the same machine address. By doing so, the hypervisor ensures the memory isolation between virtual machines.

Synthesis

In this section, we showed the different methods of system virtualization and their evolution over time from the performance perspective. We show that the role of the virtualization program changed. In the first place, the cycle accurate simulator was intended to reproduce precisely every aspect of the hardware. Later, the hardware emulator and the DBT emulator traded this precision to improve performance by only emulating the external behavior of a physical machine. Finally, the paravirtualizing hypervisor, then the HAV compatible hypervisor delegate most of the emulation work. The paravirtualizing hypervisor simply leaves this emulation, letting the guest system knowing it executes in a virtual machine. The HAV compatible hypervisor lets the processor handle most of the simple emulation operations. Both of these hypervisor thus focus on high level virtual machine management. In the next section, we describe virtual machine management issues that can occur and how existing works address them.

Related challenges

In Section 2.3, we show that the role of modern hypervisors is to implement high level resource management for the virtual machines. This contrasts with the older virtualization systems that focus on how to efficiently emulate a virtual hardware. To ease the resource management, today hypervisors usually expose simple hardware to their guests. Typically, a hypervisor exposes a set of vCPUs to the guest system. From the guest point of view, a vCPU is a device that constantly fetches and processes new instructions until the operating system stops it. From the host point of view, a vCPU is an execution thread that is arbitrarily paused, restarted or moved from a core to another one by the hypervisor scheduler.

Exposing simple hardware abstractions to the guest system has many benefits. First, there is no need to modify the guest operating system. Since the vCPU behaves exactly like a physical CPU, the operating system interacts with hardware it is used to. Second, using abstractions makes the hypervisor maintainable. Handling many types of device is a source of software complexity[3]. Providing many types of device is too. Third, by hiding the details

[3] For this reason, some hypervisors, like Xen, delegate this task to a special Linux guest

Solution	Opacity		Approach
	Ignoring layer	Ignored layer	
AQL_Sched	hypervisor	application	monitoring
Vscale	operating system	hypervisor	transparency
AASH	operating system	hardware	transparency
Bias Random Migration	operating system	hardware other guests	monitoring
	hypervisor	application	

Table 2.7: Works related to system virtualization

of resource management behind abstractions, the hypervisor improves security. Indeed, exposing the internals of the hypervisor to a malicious guest increases the attack surface.

On the other side, exposing simple hardware abstractions to the guest system also has drawbacks. As the hypervisor hides the hardware details to the operating system, it thus prevents this last to use the hardware efficiently. This opacity between the guest software and the hardware has a huge impact on application performance, as we explain in Section 2.1 and Section 2.2. Moreover, the hypervisor hides its own behavior to the guest operating system. These two software layers, the hypervisor and the operating system, both manage a set of resources. Because of the opacity between these two layers, they can take contradictory decisions that cause performance drops.

In Section 2.2, we show that some policies rely on the end user application behavior to optimize the memory placement. From the hypervisor point of view, a virtual machine is a black box. The hypervisor only sees a set of vCPUs that sometimes access virtual devices or cause a `VMEXIT`. The hypervisor has no access to the end user applications and is thus unable to know their behavior. This other kind of opacity likewise impacts the application performance by preventing the hypervisor to take accurate resource management decisions.

Table 2.7 enumerates the works we present in this section. For each of them, we describe the issue it addresses, and emphasize the opacity that causes this issue. We characterize the opacity by its ignoring part and its ignored part. The ignoring part is the software layer that lacks information about the ignored part to take efficient decisions. We also indicate what approach these works use, either a monitoring or a transparency approach.

Monitoring In this approach, the hypervisor uses a set of hardware of software counters to capture the guest behavior. This approach does not involve the guest. The hypervisor then uses the metrics it has collected to take decisions.

Transparency In the transparency approach, the hypervisor and the guest operating system exchange information to weaken the opacity. Both these layers can exchange about their own internals or about another layer they have access to. For instance, the hypervisor can expose the hardware details to the guest operating system.

AQL_Sched

A modern hypervisor is responsible of high-level resource management for its virtual machines. One of the resources it manages is the CPU time. The hypervisor schedules the vCPUs of the virtual machines on the physical cores. A hypervisor scheduler roughly has the same goals than an operating system scheduler: ensure fairness[4], prevent starvation and maximize responsiveness. The vCPUs, scheduled by a hypervisor, are however slightly different than threads scheduled by an operating system. On the one hand, a thread behaves quite simply: it executes constantly until the end of its quantum. Optionally, a thread can also perform blocking operations. On the other hand, a vCPU executes several guest threads with their own quantum and serves virtual interrupts.

A vCPU is thus a composite execution thread i.e. it is the aggregation of several execution threads at the guest level. For this reason, hypervisors usually schedule vCPUs with larger quanta than the operating systems do with their threads. A typical hypervisor quantum length is 50 ms against 1 ms for an operating system quantum length. As Teabe et al. [66] point it, this large quantum length is beneficial to computation intensive guest applications, since it lowers cache pollution. A large quantum is however detrimental to I/O intensive ones. Indeed, an application performs an I/O by sending a request to an I/O device, then blocks. When the device has served the request, it sends an interrupt to one of the vCPUs, chosen by the guest operating system, to resume the application. If the vCPU is not scheduled, the interrupt is pending. This interrupt remains pending until the hypervisor schedules the targeted vCPU. A large scheduling quantum thus lowers the guest responsiveness.

Teabe et al. propose a new scheduler for the Xen hypervisor that adapts its quantum length to the application: the AQL_Scheduler [66]. This scheduler classifies the vCPUs in three types that determine their quantum length. This classification bases on four criteria: the LLC access frequency, the LLC miss rate, the `PAUSE` instruction frequency and the number of I/O request. The `PAUSE` instruction purpose is to detect vCPUs that spinlocks. We discuss more about the virtualized spinlocks in the Vscale work. The hypervisor obtains the first two criteria with hardware performance counters, the third criterion with instruction interception and the fourth one with a software counter.

The classifier first separates the vCPUs with a high LLC miss rate from the other ones. The vCPUs of the same group are placed on the sames LLC. Inside of each group, vCPUs with a high LLC access frequency are scheduled on the same cores with a large quantum length. A high LLC access frequency indicates that the vCPU is computation intensive. Also, vCPUs with a high frequency of `PAUSE` instruction or a large amount of I/O need responsiveness and are thus scheduled on cores with small quantum length.

The AQL_Sched thus monitors the guest vCPUs to capture the high level behavior of the guest applications. Depending on this behavior: computation intensive, I/O intensive or synchronization intensive, the hypervisor schedules the vCPUs with different quantum length.

[4]The machine administrator can however attribute different priorities to different virtual machines

Vscale

As stated in the presentation of AQL_Sched, virtualized spinlocks, and actually, any synchronization between vCPUs, are a special performance concern. A guest system uses spinlocks to protect critical sections that a thread can executes quickly. Indeed, waiting on a spinlock consumes CPU cycles. Thus, synchronizing large critical sections with spinlocks would be a waste of computing power. Using spinlocks in a non virtualized kernel is safe because (i) the operating system ensures that a thread that owns a spinlock is not scheduled out and (ii) this thread executes on a core that fetches and executes instructions until told otherwise by the system. A thread that owns a spinlock is thus guaranteed to release the spinlock in short delays.

Contrary to a physical core, a vCPU may be paused by the hypervisor scheduler. As a result, a vCPU not necessarily fetches and executes instructions until told otherwise by the guest system. A guest thread executing on a scheduled vCPU can thus acquire a spinlock, then the vCPU of this thread is scheduled out by the hypervisor. If another guest thread executing on another vCPU now tries to acquire the spinlock, it has to wait for the hypervisor to reschedule the owner vCPU. Since vCPUs quanta are usually tens of milliseconds, acquiring a spinlock in a guest is potentially a huge waste of time and CPU cycles. This scenario is called the *lock holder preemption*.

This lock holder preemption problem is caused by the opacity between the guest operating system and the hypervisor. Indeed, the hypervisor usually presents a fixed number of vCPUs to the guest, independently of the total number of vCPUs that compete for execution on physical cores. The guest thus assumes a fixed amount of processing power, equally spread between the vCPUs. This assumption does not hold when the hypervisor consolidates several guests on the same hardware. When more guests compete for the same cores, the amount of processing power per guest decreases. This opacity between the hypervisor and the guest operating system leads the last to acquire spinlocks with vCPUs that are likely to be preempted.

Cheng et al. address the lock holder preemption problem with Vscale [19]. The main idea of Vscale is to expose the amount of physical CPU time to the guest system. To this end, the hypervisor does not expose a fixed number of vCPUs to the guest. Instead, it exposes a number of vCPUs proportional to the number of physical CPUs allowed to the guest. The scheduling strategy thus shifts from "having a fixed number of vCPUs, each with a decreasing execution time" to "having a decreasing number of vCPUs, each with a fixed execution time". When changing the number of vCPUs of a guest, the hypervisor notifies it. The guest operating system thus migrates the threads away from the vCPUs that are removed, or to the vCPUs that has been added.

AASH

In this chapter, we only discussed symmetric multicores, where all of the cores are equivalent. In asymmetric multicores, different sets of cores deliver different computing powers. The *fast* cores process instructions with high throughput but consume a lot of energy

while the *slow* cores process instructions more slowly but with a lower energy consumption. When both the fast and the slow cores expose the same ISA, the load balancer chooses carefully on which core to execute each thread. Executing a thread that frequently stalls the core, waiting for memory, on the fast cores causes an energy waste. Executing a thread that could benefit of high instruction throughput on the slow cores decreases the performance.

Efficiently placing threads on asymmetric architectures is addressed in several works [10, 41, 62] for native configurations. These solutions typically monitor application threads to determine the benefits of placing them on fast cores. This monitoring is made by the operating system. These solutions do not work in a virtualized configuration. The hypervisor isolates the guest operating system from the actual hardware. Specifically, the hypervisor does not expose the fast or slow characteristic of the vCPU. The guest operating system thus sees a symmetric multicore and does not make any optimization.

Kazempour et al. tackle this problem with the Asymmetry-Aware Scheduler for Hypervisor [39]. This scheduler, implemented in Xen, is aware of the asymmetric nature of the hardware. AASH uses this knowledge to expose the asymmetry to the guests. In addition, AASH uses the asymmetry awareness to ensure a fair sharing of the fast cores among the guests, but this optimization is out of this document scope. Exposing a particular vCPU as fast for the guest is trivial. The hypervisor intercepts and emulates the instructions used by the guest to discover the properties of the vCPUs. The main difficulty is for the hypervisor to ensure that fast vCPUs always execute on fast physical cores.

On a symmetric multicore, the Xen hypervisor uses a credit-based scheduler. One credit stands for 100 μs of physical CPU time. Periodically, each vCPU receives an amount of credit proportional to its priority. The global amount of given credits for a period is proportional to the amount of physical core. When a vCPU executes, it consumes its credits. The scheduler selects the next vCPU to execute by looking the one that has the largest amount of credits.

On an asymmetric multicore, AASH uses two types of credits. The fast credits stand for fast CPU time while the slow credits stand for slow CPU time. Only a vCPU with fast credits can execute on a fast core. Both fast and slow credits are distributed periodically. The global amount of given fast credits (resp. slow credits) for a period is proportional to the number of fast cores (resp. slow cores). AASH chooses which vCPUs receive fast credits in a round-robin fashion. Each period, N vCPUs marked as fast receive fast credits, with N being the number of fast cores. Once a vCPU has consumed all of its fast credits, AASH migrates this vCPU to a slow core. This vCPU then only receives slow credits until all the other fast vCPUs have received fast credits.

By exposing fast vCPUs that effectively execute on fast physical cores, the hypervisor removes the opacity between the operating system and the hardware. This transparency allows the guest operating system to assign computation heavy application threads to the fast cores.

Bias Random Migration

Efficiently placing vCPUs on an asymmetric multicore or on a NUMA hardware are similar problems. In both cases, the main difficulty is to execute the application threads on the most appropriate physical cores. On an asymmetric multiprocessor, there is two types of cores: fast and slow. Furthermore, the operating system classifies the application threads in two types: fast and slow. The scheduler reaches an efficient placement when the fast threads execute on the fast cores. On a NUMA architecture, their are several performance factors. The data locality and the memory access balance are some of them, as we explain in Section 2.2. The contention on LLC is another performance factor. An efficient vCPU scheduler for NUMA hardware thus tries to optimize all of these performance criteria.

Several works [70, 67, 20] address the problem of efficient vCPU scheduling on NUMA hardware. We present one of them, the Bias Random vCPU Migration [56] system that characterizes several performance factors with a unique metric. This metric is called the *uncore penalty* and is the number of cycles a vCPU stays stalled because of memory accesses. The uncore penalty captures the cost of an inefficient memory placement whether it be a locality or balance issue.

The uncore penalty metric actually measures the consequence of a memory placement rather than the causes of an inefficient placement. When a vCPU executes in a given NUMA node, the BRM records the uncore penalty associated to the pair (vCPU, node). The BRM system uses these uncore penalties to decide on which node to move a vCPU when asked by the hypervisor scheduler. The node is decided randomly but with a bias: the BRM more likely chooses to place a vCPU on a node if the uncore penalty of the (vCPU, node) pair is low. Because of this bias, the BRM migrates the vCPUs on the NUMA nodes it knows the placement is efficient. Because the BRM decides randomly, it eventually explores all the pair (vCPU, node) and collect the uncore penalty metric for each of these pairs.

Like many other works related to NUMA virtualization [44, 21], the BRM system tackles an opacity problem that has several aspects. On one side, the hypervisor does not see the application behavior: which thread accesses to what memory zone. On the other side, the guest operating system is unaware of the NUMA hardware. Additionally, the guest system does not see the other guests either. These other guests generate their own memory accesses. Accounting this additional memory load is important to prevent memory contention. Like many other works related to NUMA virtualization, the BRM uses performance counters to monitor the guests. The BRM then takes placement decisions depending on the collected information.

Conclusion

The presented works address the problem caused by the virtualization opacity. The hypervisor isolates the guest operating system and applications from the hardware. Additionally, the hypervisor sees the guest as a black box and hence isolates itself from the end user application. This opacity prevents all the software layers to take efficient decisions. This situation is particularly detrimental when the hardware has a NUMA architecture. As we

		Hard drive MB/s	Context switches k/s	Memory footprint MB
Parsec	bodytrack	0	17.7	7
	facesim	0	11.7	328
	fluidanimate	0	4.2	223
	streamcluster	0	29.5	106
	swaptions	0	0.0	4
	x264	0	0.6	1129
NPB	bt.C	0	1.2	698
	cg.C	0	5.9	889
	dc.B	175	0.1	39273
	ep.D	0	0.0	49
	ft.C	0	0.3	5156
	lu.C	0	1.5	600
	mg.D	0	1.5	27095
	sp.C	0	2.0	869
	ua.C	0	37.4	483

		Hard drive MB/s	Context switches k/s	Memory footprint MB
Mosbench	wc	0	3.9	16682
	wr	1	5.2	19016
	wrmem	5	7.5	11610
	pca	0	0.3	5779
	kmeans	0	0.1	4178
	psearchy	54	0.8	28576
	memcached	0	127.1	2205
X-Stream	belief	234	0.0	12292
	bfs	236	0.0	12291
	cc	249	0.0	12291
	pagerank	240	0.0	12291
	sssp	261	0.0	12291
YCSB	cassandra	16	10.7	1111
	mongodb	184	14.6	1092

Table 2.8: Behavior of the applications

show in Section 2.2, knowing the hardware and application details is essential to (i) implement efficient NUMA policies and (ii) choose what NUMA policy to apply. In particular, the virtualization opacity prevents to use existing solutions that work in native configurations.

To break the opacity, the presented works rely on two methods. In the first method, the hypervisor monitors, with hardware or software counters, the behavior of the end user application. After collecting enough information, the hypervisor takes a placement decision. In the second method, the hypervisor and the guest communicate and exchange information so one of the software layer has enough information to take decisions. In the remainder of this thesis, we show that both of these methods can be used to implement already existing NUMA policies.

2.4 Software settings

In the remaining of this document, we frequently evaluate the impact of different configurations by measuring their impact on application performances. We describe here the applications we use for those evaluations.

We take 29 applications from 5 different benchmark suites. The first suite is Parsec 2.1 (precompiled version) which contain applications working on complex data structures such as fluid simulation, vertice clustering or video encoding. The second suite is Nas Parallel Benchmark 3.3 (openMP version) which contain applications performing a lot of computation over simple data structures. Conjugate gradient computation and Block tri-diagonal solvers are good examples. The third suite is the Mosbench containing Metis applications (in memory map reduce), a parallel search engine and `memcached`. These applications allocate large amount of memory and some of them have a high allocation rate. For these reason they use a tuned memory allocator: Streamflow [58]. The fourth suite is a set of X-Stream applications. X-Stream is a framework designed to process graph formatted data too large to fit in memory and thus focus on efficient disk access patterns. The fifth and last

benchmark suite is YCSB which contains scaling database applications using the disk but in a reactive way whereas the X-Stream applications do it in a batch way.

We report in Table 2.8 the application we use, the benchmark suite from which they come and for each of them, three characteristics measured on Linux with the default configuration: the hard drive usage, reported as the average MB/s of read and write, the average rate of voluntary context switch and the maximum memory footprint, reported in MB. For the different evaluations, we use Linux 3.9 along with gcc 4.6.3, libgomp 3.0 and glibc 6. For virtualized configurations, we use Xen 4.5.

2.5 Conclusion

Computers usage is evolving. From a hardware perspective, the NUMA architecture, a 30-years-old technology, is becoming common in data centers. This popularity comes with performance improvements. The cost of a remote node memory access is no more the main bottleneck, thanks to the high throughput interconnect and to the cache coherency protocols. Instead, performance in NUMA architectures are now achieved by using NUMA policies. These resource management policies take several hardware and software aspects into account and optimize memory locality, memory balance and task placement. From a software perspective, the system virtualization is now mainstream in cloud computing. This additional software layer changed from slow simulations of hardware wires to low overhead event interception systems. The performance of virtualized applications is no more conditioned by the hypervisor ability to emulate the hardware at a low level. Instead, the resource management policies of the hypervisor is now the main performance factor.

These two technologies, NUMA architectures and system virtualization have already been studied by many works, independently. They now tend to be used together in the context of the cloud computing. Virtualized NUMA architectures is a relatively new topic. As the virtualization isolates the guest from the hardware, the traditional NUMA policies do not work. The existing works follow one of these two approaches: (i) use new hypervisor specific policies relying on guest monitoring, (ii) expose the hardware details to the guest so this last can use traditional policies. In this PhD thesis, we explore a third approach: shifting the traditional NUMA policies from the guest to the hypervisor.

CHAPTER 3

THE WELL-KNOWN BOTTLENECKS

In this work, we mainly focus on NUMA related overhead of virtualization. However, real world application are generally affected by several factors, some of them are well known. In order to highlight the NUMA effects on virtualization, we try to remove already known performance limitation factors. As removing these other factors is not the goal of this thesis, we only provide quick fixes, unsuitable for production uses, but convenient for our main work.

We identify two bottlenecks affecting our applications: the virtualized I/O overhead and virtual IPI overhead, that we successfully remove. We also study a NUMA related performance factor, the task load balancing, which surprisingly has only a low impact on application performance.

3.1 The Virtualized I/O Overhead

In this section, we provide some background on the I/O subsystems available in Xen. These techniques are not specific to Xen and can be found in the other mainstream hypervisors. The implementation design of Xen however may lead to additional costs. The causes of these costs are also explained in this section.

Hardware Emulation

For isolation purpose, the hypervisor isolates the guest operating system from I/O devices. Indeed, in order to allow several virtual machines to use the same device, the hypervisor intercepts guest requests and multiplexes the accesses. Besides this aspect, even devices dedicated to a single virtual machine must be isolated. This is typically the case for DMA devices. These devices take a memory address to know where to read or write in memory.

In common architectures, the device controllers are connected to the main memory without any Memory Management Unit in between them. As a result, the device controllers use untranslated memory addresses. For this reason, the hypervisor cannot use paging techniques (see Section 2.3) to isolate DMA devices. Hence, the hypervisor intercepts guest accesses to DMA devices.

As stated in Section 2.3, a simple technique is to make the hypervisor emulate the desired controller. When the guest sends a command to the controller, the hypervisor intercepts the privileged I/O instructions and updates the state of the virtual controller. Eventually, the hypervisor translates the intercepted memory address and sends a request to the actual device controller. While this approach is straightforward, intercepting privileged instructions and maintaining a state for an emulated device are expensive.

The paravirtualization technique avoids this overhead by letting the guest to directly call the hypervisor, asking it to perform the I/O. The hypervisor then only checks if the guest has sufficient permissions i.e. it is allowed to use the device and the memory address it gives is correct. If the guest request is valid, the hypervisor translates the memory address and transmits the request to the actual device controller. These two techniques still require the hypervisor performs a software translation of memory addresses, thus adding an overhead to each I/O operation.

Moreover, the way Xen performs I/O operations adds an overhead. Besides the user virtual machines, called *domU*, Xen maintains a privileged virtual machine called *dom0* for administration purposes. Instead of implementing its own I/O drivers, Xen uses the drivers of the dom0 to perform I/O operations[1,2]. While this technique allows Xen to use existing drivers, it also forces it to schedule the dom0 at each I/O request.

The address translation and dom0 scheduling thus adds an important overhead. On the machine we use for our experiments, we measure it with a microbenchmark performing a read of 4 KiB from the disk (using the POSIX `O_DIRECT` flag to avoid caching). This read takes 74 μs on a native Linux but 307 μs in a virtual machine with the paravirtualized driver.

The I/O Memory Management Unit

The interception of I/O requests and the software translation of memory addresses necessarily add overhead to I/O operations. The hypervisor however must intercept and translate I/O requests to guarantee the guest isolation. Otherwise, the guest could command the I/O controllers to write to arbitrary memory locations without any control of the hypervisor. Modern hardware provides various ways to address this issue. The machine we use for our experiment implements the I/O Memory Management Unit (IOMMU) [6]. Just like the usual CPU MMU, the IOMMU performs an automatic translation of addresses. While the CPU MMU, implemented in the processor, the IOMMU is located in an I/O controller[3], as

[1] The dom0 is usually a paravirtualized Linux especially configured for this usage.

[2] For heavy I/O configurations, this work can be delegated to dedicated virtual machines called *backend domains*

[3] We call I/O controller the circuitry controlling to which device the data is sent or received.

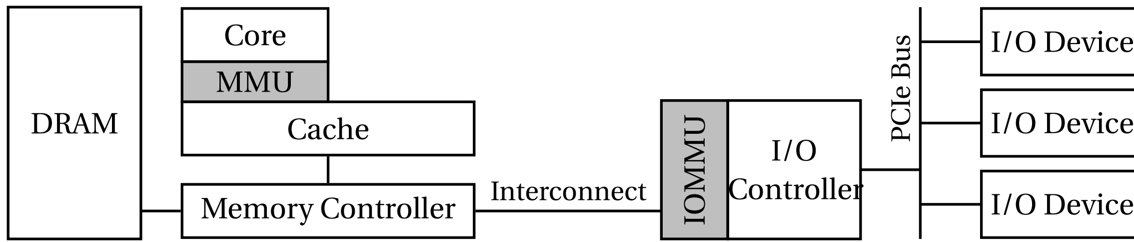

Figure 3.1: A topology example of a system with an IOMMU. Both MMU and IOMMU have their own TLB they refill by reading the page table in DRAM.

illustrated in Figure 3.1. The IOMMU thus translates the memory addresses used by I/O devices. In the remainder of this document, we use the term MMU to designate the CPU MMU and use the specific term IOMMU otherwise.

At the guest boot time, the machine administrator can choose to dedicate a device to the guest. In this case, the hypervisor configures the IOMMU to use a page table located in the main memory. This page table has the same structure than a usual MMU page table[4]. Moreover, the IOMMU and MMU page tables use the same structure bits i.e. address of the next page table level and presence bit. Because of this specificity, the hypervisor can configure both the MMU and the IOMMU with the same page table. Additionally, the IOMMU and MMU use different bits to encode permissions. The MMU reads permissions on the least significant bits and ignore the most significant bits, while the IOMMU does the contrary. The hypervisor can thus grant different permissions to vCPU and devices. Table 3.1 and Table 3.2 indicate the layout of MMU and IOMMU entries.

The IOMMU also uses a table to remap device initiated interrupts. This table associates vCPU ids to pCPU ids. The hypervisor configures this table and updates it after each vCPU migration so the devices send interrupts to the appropriate pCPUs. This automatic interrupt redirection allows the guest system to implement its own interrupt policies without any hypervisor interception. The IOMMU also implements several other features not discussed here. Also, like a regular MMU, the IOMMU uses a TLB to cache already performed translations. The hypervisor can flush this TLB when needed, for instance after an entry update.

An important difference between the IOMMU and the usual MMU is that the IOMMU is located outside of the processor. As a consequence, the IOMMU cannot work synchronously with the CPU. This becomes important in the case of a page fault. When the MMU fails to translate an address, it triggers a fault on the CPU. The CPU then resolves the fault or kills the faulting task, then resumes to normal execution. The IOMMU is located far from the faulting CPU, possibly on a remote NUMA node. The communication between the IOMMU and the CPU is thus too expensive for synchronous faults. Instead, the IOMMU registers a page fault event in a memory buffer and notifies the CPU with an asynchronous interrupt.

[4] The IOMMU page table however has a configurable depth between 1 and 6 levels, while the MMU page table has 4 levels on modern hardware

Table 3.1: Format of a page table entry for the MMU. The Page Address indicates the physical address of the next level, the PR bit indicates if this entry is valid, the other bits indicate permissions or caching behavior.

Table 3.2: Format of a page table entry for the IOMMU. The Page Address indicates the physical address of the next level, the PR bit indicates if the entry is valid, the Next Level allows more flexible table topologies, the other bits indicate permissions, coherency requirements or are used for ATS protocol.

The hypervisor is responsible for checking the event memory buffer on the interrupt reception. During this time, the IOMMU does not stop and simply reports an I/O error to the guest.

Intel implements an alternative technology called SR-IOV [38, 61]. An SR-IOV enabled device can emulate one or more virtual versions of itself. During the guest boot, the hypervisor configures the SR-IOV device to create a new virtual configuration space. The hypervisor then gives this virtual configuration space, whatever its nature, to the guest. The SR-IOV device then handles the memory translation and guest multiplexing by itself. This technology has more requirements than the IOMMU technology since legacy devices cannot be used. Since this technology is not available in our experimental hardware, we focus on the IOMMU approach.

Xen can use an available IOMMU by using the PCI-passthrough driver. The machine administrator can configure a guest to use some devices with this driver. In this case, the hypervisor gives the guest an exclusive access to these devices and configures the IOMMU. The guest thus directly accesses the devices without any hypervisor interception. With this setting, we have measured that reading a 4 KiB block takes 186 μs (compared to the 307 μs of the paravirtualized driver and the 74 μs of a native configuration). Note that the larger the amount of bytes read, the lower the overhead caused by virtualization. This is explained by the fact that when the number of bytes to read increases, the time it takes to start a DMA transfer becomes negligible compared to the time it takes to perform the transfer itself.

The PCI-passthrough driver is relatively restrictive. While the IOMMU can associate devices to virtual machines at the device granularity, the PCI-passthrough driver associates

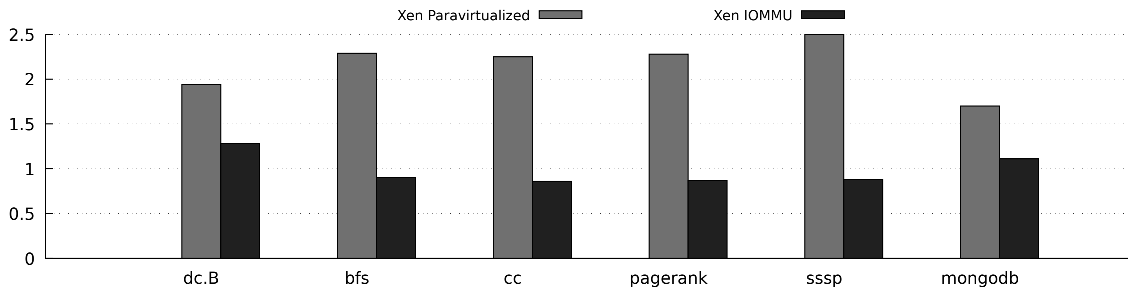

Figure 3.2: Relative overhead of various configurations compared to the best known native configuration for I/O applications. (**lower is better**)

devices to virtual machines at the PCI bus granularity. All the devices belonging to a PCI bus must be exclusively associated to a unique guest. Fortunately, as we describe in Section 2.2, the machine that we use has two PCI buses. In our experiment, we reserve a PCI bus for a domu virtual machine. The dom0 uses the other PCI bus.

Evaluation of IOMMU

To evaluate the impact of virtualized I/O on real-world applications, we execute the I/O intensive applications of our benchmark suite, namely dc.B, mongodb and the x-stream applications: bfs, cc, pagerank and sssp, on different configurations. The configuration 1 is the best native configuration we know on Linux (see Section 5.2). The configuration 2 is the default Xen configuration with the paravirtualized driver. The configuration 3 is the default Xen configuration with the PCI-passthrough driver. We compare the execution times of the applications under these configurations. The overhead of configuration 2 (Xen paravirtualized) against configuration 1 (Linux) is the cost of the virtualization, including I/O emulation. The overhead of configuration 3 (Xen PCI-passthrough) against the configuration 1 (Linux) is the cost of the virtualization with no I/O emulation. Hence, the overhead difference between configuration 2 and configuration 3 is the cost of the I/O emulation.

Figure 3.2 shows the results of the evaluation. We observe that all the evaluated applications in configuration 2 (Xen paravirtualized) have an overhead above 1.5 compared to configuration 1 (Linux), with a maximum overhead of 2.5 for sssp. With the configuration 3 (Xen PCI-passthrough), we observe two behaviors. For dc.B and mongodb, the overhead falls below 1.3, which confirms the microbenchmark result: using the IOMMU makes the I/O operations faster for the guest. For the x-stream applications, the overhead falls below 1, down to 0.86 meaning that the virtualized version is faster that the best known native configuration.

We have no definitive explanation of this counter-intuitive result, since the virtualized I/O performance is not the goal of this thesis. We however propose the following hypothesis: on a native configuration, Linux allocates DMA buffer as a contiguous region in memory, thus from a single NUMA node. Then when the DMA controller performs the read or write,

it accesses to this single NUMA node, saturating the interconnect link. This saturation may be detrimental nonetheless for the I/O operation itself, but also for the other access to this node memory. On the other side, when Xen uses the IOMMU, the DMA buffer is spread across several NUMA nodes. The load is thus balanced on several nodes, causing less saturation. This difference can explain the speedup of Xen compared to Linux.

3.2 The virtualized IPI overhead

As we describe in Section 2.1, an Inter-Processor Interrupt is a special kind of interrupt. A core asynchronously notifies another core that an event happened. The IPI by itself does not carry any information about the event. Usually, the sending core fills a dedicated memory location with information about the event. The receiving core reads this information when handling the IPI. In this section, we focus on how an IPI is implemented, both in a native and in a virtualized configuration. We see how the implementation differences may lead to performance overheads and how we remove them.

Usage of the IPI

Sending an IPI is a way for a core to asynchronously notify a remote core of an event. Operating systems use this mechanism to implement blocking synchronization on multicores. When a thread tries to acquire a blocking lock, for instance a `pthread_mutex`, it checks with an atomic operation if another thread already holds the lock[5]. If so, the acquiring thread registers itself in a waiting queue associated to the lock, then the operating system schedules the thread out. Additionally, if there is no other thread to execute on a core, the operating system puts it in halt mode. In halt mode, the core stops to fetch new instructions. This reduces the energy consumption of this core.

When a thread releases a blocking lock, it checks if there are other threads registered in the waiting queue of the lock. If so, it notifies the operating system to wake up one of them. The operating system then selects the next thread to acquire the lock and on what core to execute it. We call this core, the selected core. The selected core may be a different core than the releasing core. In this case, the releasing core sends an IPI to inform the selected core to schedule the selected thread. Additionally, a core in halt mode automatically wakes up when receiving an interrupt.

The IPI mechanism is a critical operation in blocking synchronization. Indeed, every time a thread transfers a lock ownership to another thread on a remote core, it sends an IPI. Moreover, neither the releasing thread nor the waiting threads execute while the IPI has not been sent and handled. The execution time of the IPI sending sequence thus has a large impact on applications that heavily use blocking synchronization.

[5] The pthread glibc implementation is actually more complex, see [64].

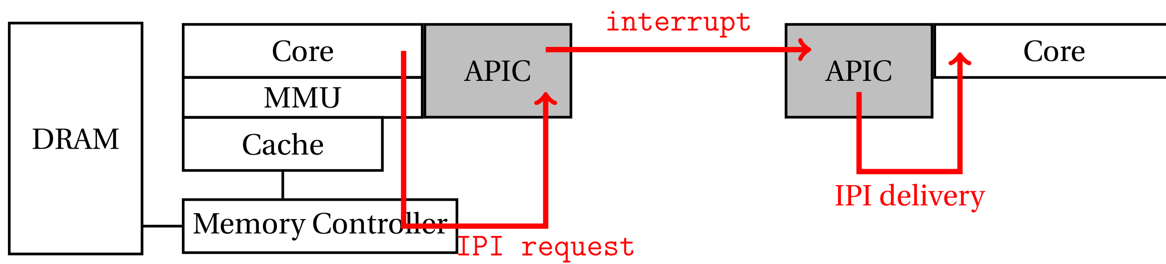

Figure 3.3: Architectural view of an IPI sending.

Implementations of IPI

On the x86 architecture, each core has an associated Advanced Programmable Interrupt Controller (APIC). This device is responsible for delivering interrupts to the core. When a device sends an interrupt to a core, this interrupt is received by the APIC associated to this core. Upon an interrupt reception, the APIC decides either to wait or to deliver the interrupt to the core i.e. making the core handle the interrupt. Typically, an APIC delivers the interrupt to its core only if the core is not already handling a higher priority interrupt. In this case, the interrupt is pending until the core gets ready to handle it.

The APIC is a memory mapped device. At boot, the operating system writes a physical memory address in a dedicated register (the APIC Base Address Register[5]). Then, the memory controller forwards all read and write requests in the specified memory address range to the APIC[6]. This mapped memory range has a well known layout. The operating system thus configures the APIC of a core my making this core writing at the appropriate offset in this memory range.

In addition to the interrupt delivery, the APIC also enables its associated core to send interrupts to other cores. A core sends an IPI by writing an IPI request in the mapped memory area (at the Interrupt Command Register offset). The IPI request is a 64 bits message that contains the id of the target core. An IPI request also contains properties like the priority of the message or the delivery mode. We do not discuss these aspects of the IPI. When the core writes the IPI request at the appropriate memory location, the local APIC sends an interrupt to the destination APIC. The destination APIC eventually delivers this interrupt to the destination core. Figure 3.3 illustrates the multicore architecture with the APIC and the whole process of IPI sending.

In a virtualized configuration, the hypervisor does not allow the guest to directly access the APIC. Indeed, the IPI request indicates the id of the physical target core, whereas a guest only sees ids of vCPU. Xen provides a paravirtualized APIC interface to the guest. The guest operating system configures a virtual APIC with hypercalls. Specifically, the guest operating system sends IPI with a dedicated hypercall that transmits an IPI request to the hypervisor.

[6]The operating system informs the cache hierarchy to forward requests for this mapped memory range, directly to the memory controller

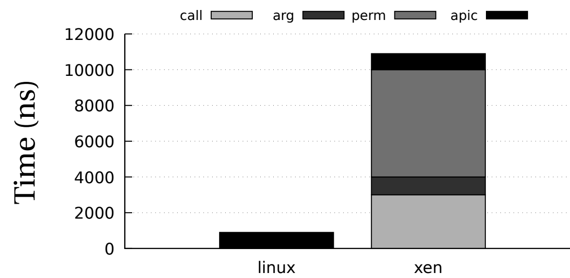

Figure 3.4: IPI cost repartition

On modern hardware, the processor provides a feature called Advanced Virtual Interrupt Controller. The hypervisor can use this feature to define a regular memory page as fake APIC mapped memory region, called backing page. Every read or write performed by the guest on the backing page is intercepted by hardware and transmitted to the hypervisor.

In the first place, we want to measure the overhead of using a virtualized version of the APIC instead of the native one. To this end, we instrument the Linux kernel and the Xen hypervisor. In each of them, we measure the execution time of the functions responsible for sending an IPI. We report these measurements in the Figure 3.4. We found that the IPI sending can be roughly partitioned in four parts. The first part is the one shared by both the native and the virtualized version: the write in hardware mapped memory which lasts 1 μs. This first part is also the total duration of the native operation.

The virtualized version of the IPI sending has three additional steps. The first one is the hypercall that lasts 2.5 μs. The hypercall is an expensive operation, far more expensive than a system call. Indeed, the hypercall makes the processor switching from guest mode to host mode. During this switch, the processor saves the guest state to memory and loads the host state. This save and load operation represents a transfer of about 800 bytes. Moreover, the processor performs the reverse operation when switching back to guest mode, at the end of the hypercall.

The second additional step is the hypercall argument copy, accounting for 1 μs. In Xen, the hypervisor and the guest do not share the same memory space. As a consequence, the Xen hypervisor cannot access directly to a guest memory region. Instead, Xen maps the memory region that contains the hypervisor arguments to a dedicated address, copies the arguments in a buffer, then unmaps the guest region. This whole process causes expensive TLB misses.

Finally, the most expensive additional step is the permission check, which lasts 6 μs. Indeed, the hypervisor guarantees that the guest operating system cannot perform illegal operations, like sending an IPI to another guest. Additionally, if the destination vCPU is not scheduled, Xen cannot deliver the virtual IPI. In this case, the hypervisor schedules the destination vCPU (under certain conditions [72]) and injects the vIPI in it. Scheduling a vCPU is also an expensive operation.

As a result, the virtualized version of IPI sending takes 10 μs more than for the native version, making it 11 times slower. An overhead of 10 μs may seem negligible. However, as we explain in Section 3.2 the operating system uses IPI to wake up waiting tasks. This additional time is thus located on the critical path of the application. We ensure an overhead of 10 μs has perceptible impact on performance with a microbenchmark. We modify the Linux kernel to add an artificial overhead of 10 μs on the native version of IPI sending[7]. With this modified version of Linux, we execute the facesim application in a native configuration. As we show in Section 2.4, facesim makes about 11700 voluntary context switch per second. We found these context switches are induced by blocking synchronizations. We observed an overhead of 50% compared to the default version of Linux. This result is a good indication that these 10 μs impact the overall performance of an application.

The libactive library

The virtualization overhead for the IPI mechanism may affect the application performance. Ding et al. [29, 63] already report this issue and propose to solve the problem with a complex algorithm which can handle multiple consolidated workloads. For the sake of simplicity, we use a more straightforward approach. We redefine the `pthread` synchronization primitives of `pthread_mutex` and `pthread_cond` to spin instead of blocking. By this mean, we ensure the cores communicate through memory instead of sending IPI. Contrary to the IPI, a memory access has no virtualization overhead. Thus, a spining implementation of the `pthread` primitives should not suffer any performance issue in a virtualized configuration.

We implement these primitives as a shared library, called libactive. When starting the benchmark application, we preload the libactive[8]. To avoid cache line contention issues, we use the MCS algorithm [49] to implement the `pthread_mutex` primitives. We also use an extension of the MCS algorithm for the `pthread_cond` primitives, with a special care for the `pthread_cond_broadcast` performance.

Figure 3.5 shows a typical usage of `pthread_cond`. At line 2, the waiter thread takes a lock. At line 3, the waiter thread checks if some condition to be satisfied. Checking if the condition is satisfied usually is a critical code protected by the lock taken at line 2. If the condition is not satisfied, the waiter thread releases the lock and blocks at line 4. The notifier thread at lines 10 and 11 sporadically wakes up all the threads waiting at line 4. When a waiter thread wakes up, it acquires the lock that it previously released and checks the condition again. When eventually the condition is satisfied, the thread do some useful work at line 5 and releases the lock at line 6.

When the notifier thread uses `pthread_cond_broadcast` at line 11 to wake up all the waiting threads, these awaken threads compete to acquire the same lock. We observed that a naive implementation of `pthread_cond_broadcast` causes a high contention on the lock. This contention has a negative impact on the application performance.

[7]We use an empty `while` loop to this end.

[8]The symbols of our library override the symbols of the default `pthread` library at run time.

```
1  void waiter_thread(void) {
2    pthread_mutex_lock(&mutex);
3    while (condition_not_satisfied())
4      pthread_cond_wait(&cond, &mutex);
5    usefull_work();
6    pthread_mutex_unlock(&mutex);
7  }
8
9  void notifier_thread(void) {
10   while (something_happens())
11     pthread_cond_broadcast(&cond);
12 }
```

Figure 3.5: A typical usage of the `pthread_cond` primitives

We provide an efficient implementation. When a thread executes the `pthread_cond_wait` function, it inserts itself in a queue associated to the condition variable before to release the lock. When a thread executes the `pthread_cond_signal` (resp. `pthread_cond_broadcast`) function, this thread moves one (resp. all) thread from the condition queue to the queue of the lock to acquire[9]. If the lock is not already hold, the first thread of the queue is woken up.

We observe that the amount of context switch for facesim drops to 0 when we use the libactive. This confirms that the libactive implementation effectively prevents the application threads to block. As a side effect, our implementation is more effective, in a single application context, than the standard implementation. Indeed, using busy waiting avoids entering in kernel mode and performing scheduling, reducing the synchronization latencies. For this reason, when we compare native and virtualized configurations, we ensure that either both or none of the configuration use the libactive.

This solution however has some limitations. First, the libactive is unsuitable for configurations with time sharing. Indeed, a thread that uses the libactive waits in a spin loop. Thus this thread wastes CPU cycles instead of releasing the processor and then, prevents other ready threads to execute. Second, the spining only occurs on pthread based synchronization. Applications synchronized with other methods, like openMP based applications, or network intensive applications still block and thus may use IPI.

Evaluation of vIPI

In this section, we evaluate the impact of virtualized IPI on real-world applications. We found that two applications of our benchmark suite heavily use the pthread synchronization primitives: facesim and streamcluster. We execute these two applications on different configurations. The configuration 1 is a native configuration with the default implemen-

[9]Actualy, several threads may wait on the same `pthread_cond` but on different `pthread_mutex`. Our implementation dispatches them correctly.

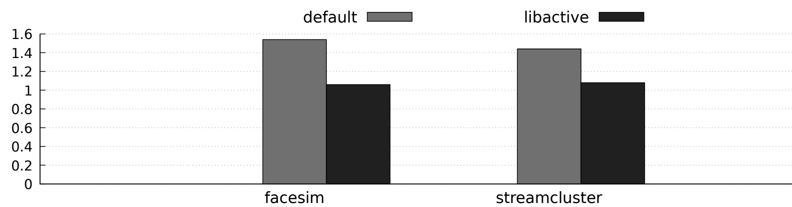

Figure 3.6: Relative overhead of various configurations compared to the best known native configuration for pthread synchronization intensive applications.

tation of pthread. The configuration 2 is a Xen configuration with the default implementation of pthread. The configuration 3 and 4 are the same configurations that 1 and 2 but with the libactive implementation of pthread. For fairness, all of these implementations use the round-4K NUMA policy (we describe how we implement the round-4K policy in Xen in Section 5.1).

We compare two overheads. The overhead of configuration 2 (Xen pthread) against configuration 1 (Linux pthread) is the cost of the virtualization, including virtualized IPI. The overhead of configuration 4 (Xen libactive) against configuration 3 (Linux libactive) is the cost of the virtualization with no virtualized IPI. Hence, the difference between these two overheads is the cost of the virtualized IPI.

Figure 3.6 shows the result of the evaluation. We observe that with the default implementation of pthread, the virtualized applications have an overhead above 1.4 compared to their native version. With the libactive implementation however, the overhead drops below 1.1 for both the applications. This evaluation confirms that virtualized IPI cause an overhead for synchronization intensive applications.

3.3 The Xen load balancer

As the number of available cores of a physical machine grows, balancing the load among these cores becomes more and more challenging. This is true for an operating system, like Linux, which balances the threads across available cores [46] and it seems reasonable to think it is also true for a hypervisor, like Xen, which balances vCPUs across available pCPUs. Writing an efficient load balancer for a hypervisor appears as even more difficult because of the double blind decision making: the guest operating system moves threads between vCPUs to balance the load but the hypervisor then moves the vCPUs between the pCPUs with the possibility to cancel out the guest work.

A common practice when working on systems performance is to pin the threads or the vCPUs to avoid performance variations. This practice lowers the variation of execution times. However, it also disables potential performance improvements that the load balancer provides. In this section, we evaluate how much the pining strategies applied to operating system and hypervisor impact the application performances. To this end, we designed a tool: the `libpin`, a pining library to override system and application pining decisions. First, we

describe in this section how the Linux and Xen load balancers work. Then we present simple pining strategies and how we implement them in the libpin. Finally, we evaluate these strategies.

Completely Fair Scheduler

The Completely Fair Scheduler (CFS) is the default Linux placement policy. It balances the load of the ready threads among all the physical cores [46]. Linux favors locality by considering the machine as a tree, in which a node represents a level of the topology (e.g., machine level, NUMA level, core level). Linux balances the load by starting the balancing algorithm at the lowest level of the hierarchy, which ensures that the threads tend to stay on their NUMA nodes. Moreover, when a thread wakes up another thread because of a synchronization, the awaken thread is placed on a neighbor, which ensures that threads that synchronize often also tend to stay on the same NUMA node. When Linux executes as a guest, it only views a flat hardware topology, and CFS can thus not enforce NUMA locality.

Credit Scheduler

The Credit Scheduler is the default Xen placement policy. It also balances the load among all the physical cores [66]. During the virtual machine boot, Xen eagerly allocates the needed physical memory, trying to pack this memory and the virtual machine vCPUs on the minimal number of underloaded NUMA nodes by reserving a pCPU per vCPU [71]. This minimal number of NUMA nodes forms the *home-nodes* of the virtual machine. Xen favors locality by scheduling the vCPUs on the virtual machine's home-nodes. In case of overloaded home-nodes, Xen schedules the vCPUs on any NUMA node.

The libpin

To measure the impact of the Linux and Xen load balancers, we define the following pining strategies:

core each execution thread is pined to exactly one execution unit. The load balancer has no effect on the execution thread.

node each execution thread is pined to every execution unit of a same NUMA node. The load balancer can thus balance the load inside a NUMA node but can never move an execution thread between two different nodes.

all each execution thread is not pined and the load balancer thus moves the execution threads on any execution unit.

These strategies (core, node, all) may be used on native Linux, referred as the `tc`, `tn` and `ta` strategies. Figure 3.7 illustrates the `tc`, `tn` and `ta` strategies.

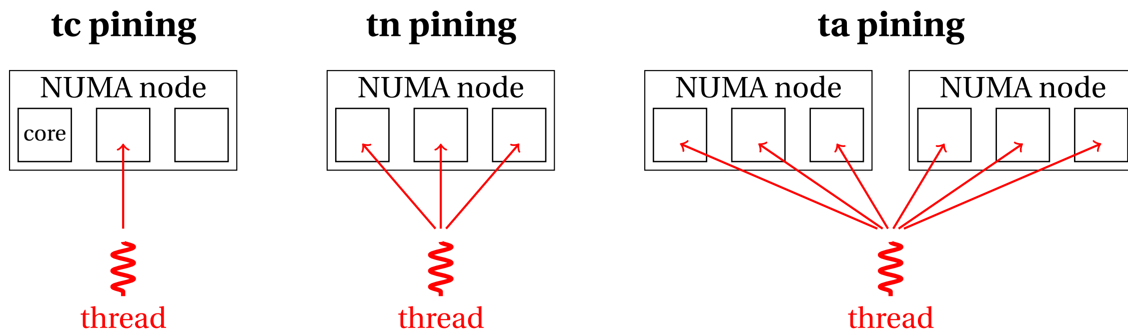

Figure 3.7: The different thread pining strategies that we evaluate. The red arrows indicate on what core a thread can execute for each pining strategy.

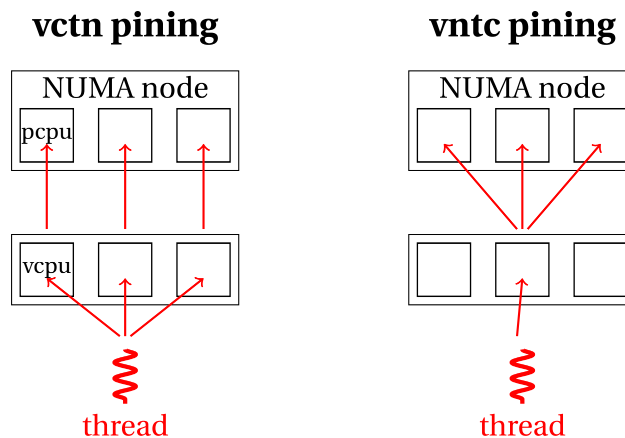

Figure 3.8: Some thread/vCPU pining strategies that we evaluate. The red arrow indicate on what vCPU a thread can execute and on what pCPU each vCPU can execute.

We can also apply the three strategies (core, node, all) to the vCPUs, in Xen. The complete pining strategy is thus a combination of the vCPU pining at the host level and the pining of the threads at the guest level. As a result, there is a total of 9 strategies combinations. We refer to these strategies combinations as $\mathtt{v}\alpha\mathtt{t}\delta$, where α is the strategy used to pin the vCPUs and δ is the strategy used to pin the guest threads. For instance, the strategy `vctn` indicates a configuration where each vCPU is pined to exactly one pCPU but each guest thread is pined to all the vCPUs running on a same NUMA node. Figure 3.8 illustrates the `vctn` and `vntc` strategies.

The strategy `vatn` has a special meaning. Indeed, with this strategy, the vCPUs are free to move between NUMA nodes. Since the guest operating system does not see the NUMA topology, it cannot ensure its threads are not moved between physical NUMA nodes. Instead, the `vatn` strategy means the vCPUs are free to move on any physical CPUs and the guest threads are restricted to arbitrary disjoint groups of vCPUs having the same size than the amount of core in a NUMA node.

Pining the vCPUs of a virtual machine is simple. The hypervisor allocates the vCPUs of a virtual machine during the guest boot. The amount of vCPU does not change until the virtual machine halts. Pining the threads of an application is more difficult. Some applications create new threads during the execution. This is the case for x264. Additionally, some applications use the `sched_setaffinity` system call to explicitly pin their threads. This is the case for some Mosbench applications. As a result, we cannot use simple tools such as the Linux `taskset` command to apply the pining strategies we defined. We overcome this issue by designing a more powerful mechanism named the `libpin`.

The libpin is, just like the libactive, a shared library that we preload when we start the benchmark application. The libpin allows the user to define a core pining order. The user defines a list of core, or core set, before the application starts. The libpin then intercepts the thread creation requests (the calls to the `pthread_create` primitive[10]). Each time the application creates a new thread, the libpin pins this new thread to one of the cores, or core sets, of the list. The libpin assigns the cores, or core sets, in a round-robin fashion. We design the list of core, or core set, to reflect the `tc`, `tn` or `ta` strategy. This pining method can cause severe load imbalance for adversary thread creation patterns. However, we found that it performs well for the applications of our benchmark.

Additionally, the libpin intercepts the application calls to `sched_setaffinity` and remap the pining request to the user defined pining. Some applications, like the Mosbench applications, define their own pining scheme. We however found that these pining schemes relies on false assumptions. For instance, the Mosbench applications assume that cores with consecutive ids belong to the same NUMA nodes. The applications use this assumption to place communicating threads on the same NUMA node. The assumption does not stand on the hardware we use: two cores with consecutive ids are never on the same NUMA node. As a result, default Mosbench applications access to the memory with a poor locality.

Evaluation of pining scheme

In this section, we evaluate the impact of the load balancer decisions on the applications performance under native and virtualized configurations. To this end, we execute several applications from our benchmark suite with different pining strategies. Each of these applications is executed with one of the 3 native pining strategies and with one of the 9 virtualized pining strategies. We try to observe if (i) some of the pining strategies perform systematically better than the other ones and (ii) there is a correlation between the best native pining strategy and the best virtualized pining strategy. Additionally, for the virtualized strategies, we want to know if the threads pining has the same impact on performance than the vCPUs pining.

We report in the table 3.3 a set of 15 metrics per application. For each application, the 3 upper cells indicate the overhead of the thread pining strategies on Linux as compared to the best Linux strategy. For instance, for ua.C, pining the threads by core (resp. by node) is 20% (resp. 17%) slower than not pining them on Linux. The 3 cells below indicate the

[10] Our version of OpenMP also uses the `pthread_create` primitive to create tasks.

Linux	thread pining	core			node			all		
Xen	thread pining	core			node			all		
	vcpu pining	core	node	all	core	node	all	core	node	all
bodytrack		2%			0%			5%		
		21%			**13%**			0%		
		0%	0%	1%	**11%**	9%	0%	1%	0%	4%
facesim		0%			0%			6%		
		0%			2%			0%		
		6%	6%	0%	4%	4%	0%	2%	0%	2%
fluidanimate		0%			3%			0%		
		0%			0%			1%		
		0%	0%	0%	0%	1%	0%	1%	0%	0%
streamcluster		0%			0%			0%		
		0%			2%			0%		
		0%	1%	1%	0%	1%	1%	3%	0%	0%
swaptions		7%			0%			0%		
		0%			0%			0%		
		0%	0%	0%	0%	0%	0%	0%	0%	0%
x264		**56%**			0%			0%		
		61%			7%			0%		
		0%	7%	7%	3%	0%	0%	**15%**	0%	3%
bt.C		0%			**64%**			**57%**		
		0%			0%			0%		
		1%	2%	0%	3%	3%	0%	0%	0%	0%
cg.C		0%			**27%**			0%		
		8%			4%			0%		
		0%	0%	1%	4%	5%	0%	0%	1%	0%
dc.B		0%			**38%**			**41%**		
		3%			0%			2%		
		4%	1%	0%	4%	4%	0%	2%	0%	2%

Linux	thread pining	core			node			all		
Xen	thread pining	core			node			all		
	vcpu pining	core	node	all	core	node	all	core	node	all
ep.D		0%			1%			**26%**		
		0%			0%			0%		
		3%	0%	0%	1%	**32%**	0%	0%	0%	1%
ft.C		1%			4%			0%		
		0%			2%			2%		
		0%	2%	2%	0%	1%	0%	0%	1%	1%
lu.C		0%			**13%**			3%		
		0%			1%			0%		
		0%	1%	1%	1%	3%	0%	0%	0%	0%
mg.D		0%			0%			0%		
		1%			0%			0%		
		1%	1%	0%	1%	2%	0%	0%	0%	0%
sp.C		1%			2%			0%		
		1%			1%			0%		
		5%	5%	0%	6%	6%	0%	3%	1%	0%
ua.C		**20%**			**17%**			0%		
		46%			**24%**			0%		
		0%	0%	0%	**23%**	**23%**	0%	0%	0%	0%
kmeans		1%			0%			1%		
		0%			2%			2%		
		0%	4%	3%	0%	1%	0%	0%	0%	0%
psearchy		0%			**26%**			**24%**		
		0%			0%			1%		
		19%	0%	2%	2%	0%	4%	1%	0%	3%
memcached		**16%**			0%			0%		
		15%			0%			**16%**		
		18%	9%	0%	0%	3%	**17%**	1%	0%	0%

Table 3.3: Effect of pining strategies on various applications. Each slot in the array is the overhead of the corresponding strategy compared to the best strategy. Overheads larger than 10% are in bold.

overhead of thread pining strategies on Xen for the best possible vCPU pining strategy as compared to the best Xen strategy. For instance, for bodytrack, pining the threads by core (resp. by node) is 21% (resp. 13%) slower than not pining them on Xen. The 3 groups of 3 cells below indicate for each Xen thread pining strategy, the overhead of vCPU pining strategies as compared to the best vCPU pining strategy for this thread pining strategy on Xen. For instance, for bodytrack in Xen with the threads pined by node, pining the vCPUs by core (resp. by node) is 11% (resp. 9%) slower than not pining them. The percentages are made proportionally to the best possible pining strategy for the considered system (Linux or Xen). The values below 10% are in light gray.

The first observation we make is that for a large amount of application, the pining strategy has no impact: over the 18 evaluated applications, only 9 of them under Linux and 6 of them under Xen have a performance gap higher than 10% between the different strategies. As a second observation, we notice that if the pining strategy has an impact in Linux, it does not necessarily has an impact in Xen. This is the case for bt.C, cg.C, dc.B and lu.C. Finally as a third observation, we see that when the thread pining strategy is important in both Linux and Xen, the best pining strategy is the same. This is visible for the applications x264, ua.C and memcached.

This evaluation suggests that the pining strategy has only a marginal impact on the applications performance, always below 64%. When the pining strategy has an impact, very few correlation can be made between the effects on Linux and Xen. Since the overhead variation differs between Linux and Xen in 26% at maximum when using the `tc` and `vctc` strategies, which pin every execution thread to exactly one execution unit, we use these

strategies in the next evaluations.

3.4 Conclusion

This chapter describes three possible causes of performance degradation for applications executed in a virtualized environment: the overhead of virtualized I/O, virtualized IPI and the interactions between the guest and hypervisor load balancers. We show that the first two effectively decrease the application performances and we thus provide a fix for each of them: we delegate the direct control of I/O devices to the guest to remove the virtualized I/O and their overhead, and we synchronize guest threads without using IPI to remove the cost of virtualized IPI. We also show that if the load balancers interactions have an impact on application performance, this impact is limited and there seems to be no correlation between the best native and the best virtualized strategy. We thus choose to cancel every load balancer effects by pining every vCPU and every thread manually.

Previous works [65, 43, 12] tackles other problems related to virtualization. These problems are however not related to the NUMA topology and only occurs when two virtual machines or more are collocated on the same physical machine. Since this work focus on NUMA related effects on virtualization, we mainly evaluate the application performances by executing either alone on a virtual machine executed exclusively on the physical host. In combination with the fixes we describe in this chapter, this removes most of the virtualization overhead causes, only letting NUMA related overhead.

CHAPTER 4

THE NUMA BOTTLENECK

In Chapter 3, we describe some well-known virtualization bottlenecks. We also show how, with simple fixes, we improve the performance of virtualized applications. The fixes we describe already remove a large portion of the virtualization overhead: the I/O intensive applications overhead falls below 1.3 and the IPI intensive applications (based on `pthread`) overhead falls below 1.1. We can wonder if the NUMA management is a bottleneck in virtualized configurations.

In this Chapter, we evaluate the impact of the NUMA management on native and virtualized configurations. In a first part, we show how the NUMA policies impact the applications performance in native configurations. This first part helps to understand (i) how much improvement we can expect from an appropriate NUMA management in virtualized configurations and (ii) how to choose an efficient NUMA policy for a given application. In a second part, what NUMA policy the Xen hypervisor provides and how it affect the applications performance.

4.1 NUMA policies under study

This section evaluates the performance impacts of the NUMA policies presented in Section 2.2, namely the first-touch policy, the round-4K policy and the Carrefour policy. To this end, we measure the execution time of each of the 29 applications of our benchmark, presented in Section 2.4 on a Linux kernel with the different policies. We configure the applications to use all of the 48 available CPUs. We also measure various metrics with hardware counters to understand how these policies affect the applications performances.

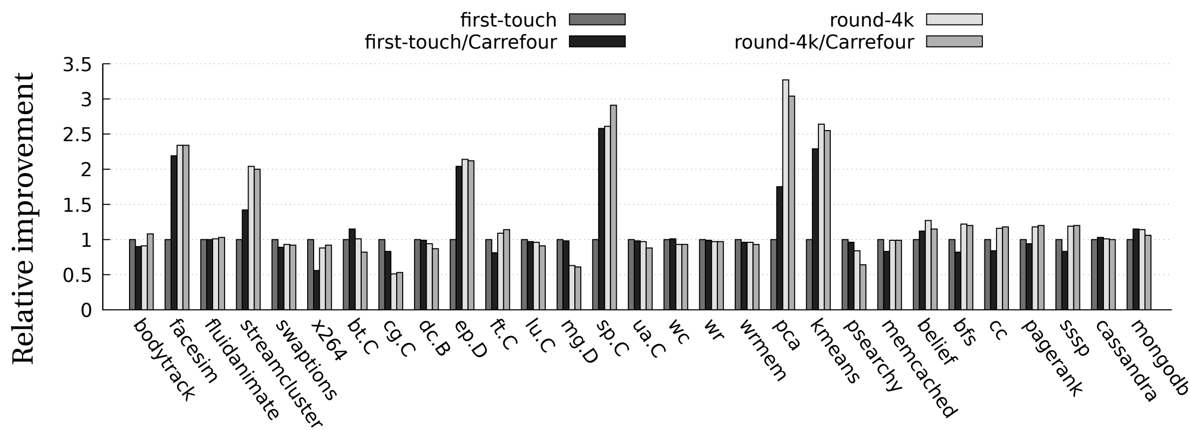

Figure 4.1: Improvement of the completion time of various NUMA policies in Linux on AMD48 with 48 threads relative to the first-touch policy (higher is better).

Comparison of NUMA policies effects

In the following experiment, we execute each application on Linux with four policies combinations: first-touch, round-4K and Carrefour associated with either first-touch or round-4K (named first-touch/Carrefour and round-4K/Carrefour). It is possible to associate Carrefour with one of the other policies because it is a dynamic policy: given an initial memory placement, provided by first-touch or round-4K, it migrates pages to improve the performance. In our setting, we use the libactive synchronization (see Section 3.2) and the `tc` pining strategy (see Section 3.3).

Figure 4.1 reports for each application, the execution time under a given NUMA policy as compared to the execution time under the first-touch policy. First, we observe that the NUMA policy has a huge impact on performance for many applications. Indeed, 17 of the 29 applications are improved by more than 25% when we compare the best against the worst NUMA policy (12 applications by more than 50% and 5 by more than 100%). We also observe that each possible combination is for some applications the one yielding the best possible performance (e.g. first-touch for `cg.C`, first-touch/Carrefour for `bt.C`, round-4K for `pca` or round-4K/Carrefour for `sp.C`).

Based on these experiment results, we make an assertion: since the best NUMA policy depends on the application, a NUMA system should provide several NUMA policies. There should be enough NUMA policies that, for each application, at least one policy is efficient with this application. This assertion leads to ask how to choose the NUMA policy to use for a given application in order to bring the best performance.

The NUMA policy selection metric

In the following experiment, we show how a NUMA policy affects the application performance. More specifically, we show how the NUMA policies modify the way the applications

use the underlying hardware. To this end, we execute the applications of our benchmark in a native configuration with either a first-touch policy or a round-4K policy. We measure two metrics with hardware performance counters: the load imbalance and the interconnect load. The load imbalance is defined as the relative standard deviation around the average number of accesses per node. The interconnect load is defined as the average of the percentage of the bandwidth used on the most loaded interconnect links during each second.[1] We do not measure these metrics with Carrefour because Carrefour already uses all the available performance counters.

Table 4.1 reports the two metrics measured with the first-touch and the round-4K policies. We classify the applications in three groups, indicated in the last column of Table 4.1. The 11 "low" applications exhibit a low memory access imbalance of less than 85% with the first-touch policy in Linux. As presented in Section 2.2, for these applications, the first-touch policy is perfect, because each thread tends to mostly access data structures that it has allocated. Carrefour tends to degrade their performance. Technically, a page mainly accessed from its node may be temporarily heavily accessed by remote nodes. In this case, Carrefour observes a temporary interconnect traffic burst and migrates the page to another node. As the remote accesses are only temporary, migrating the pages does not improve performance. The migration has, however, the consequence of degrading the memory access locality for the remainder of the run. The round-4K policy also degrades performance because it decreases memory access locality: the round-4K policy roughly multiplies by 4 the interconnect load for 7 of the 11 applications (from roughly 10% to 40%, see the column Interconnect). As a result, we have measured that the first-touch policy is only 1% slower in average than the best NUMA policy for these applications, with a worst case of 10% for `ft.C`.

At the opposite, the 13 "high" applications exhibit a high memory access imbalance of more than 130% with the first-touch policy in Linux. As presented in Section 2.2, a single thread tends to allocate the memory for the other threads. The round-4K policy prevents the large imbalance of the first-touch policy (see columns imbalance in Table 4.1). Carrefour tends to improve their performance because it improves their memory access locality (the interconnect load is large for 5 applications). As a result, we have measured that the round-4K/Carrefour policy is only 2% slower in average than the best NUMA policy for these applications, with a worst case of 7% for `pca`.

Finally, the 5 remaining "moderate" applications exhibit a moderate memory access imbalance between 85% and 130% with the first-touch policy in Linux. For these applications, the first-touch policy does not perfectly balance the load on all the nodes, but ensures a satisfactory memory access locality. Using the round-4K policy degrades performance because this policy destroys the memory access locality. The Carrefour policy is useful for

[1]The hardware counters actually give a metric which varies between 50% when the link is idle and 80% when the link is saturated. We report only the variation of the bandwidth relative to this 30% amplitude. When the machine is idle, the hardware uses 50% of the bandwidth to send hardware related commands such as link synchronization commands. Those commands can be piggy-backed on software related packets. When a link saturates, it reaches only 80% of bandwidth because each remote memory request exclusively locks the link while accessing remote components such as the remote memory controller.

	Load imbalance		Interconnect load		Imbalance level with first-touch
	First-touch	Round-4k	First-touch	Round-4k	
bodytrack	**135**%	48%	9%	8%	high
facesim	**253**%	27%	39%	16%	high
fluidanimate	**65**%	16%	18%	16%	low
streamcluster	**219**%	45%	31%	18%	high
swaptions	**175**%	180%	4%	5%	high
x264	**84**%	28%	17%	13%	low
bt.C	**89**%	8%	51%	35%	moderate
cg.C	**7**%	5%	11%	46%	low
dc.B	**45**%	19%	10%	22%	low
ep.D	**263**%	116%	48%	9%	high
ft.C	**60**%	19%	17%	46%	low
lu.C	**47**%	30%	18%	41%	low
mg.D	**8**%	1%	12%	51%	low
sp.C	**113**%	4%	43%	58%	moderate
ua.C	**5**%	7%	14%	37%	low
wc	**101**%	41%	18%	17%	moderate
wr	**110**%	57%	18%	18%	moderate
wrmem	**135**%	102%	10%	11%	high
pca	**235**%	14%	52%	41%	high
kmeans	**251**%	26%	61%	42%	high
psearchy	**19**%	8%	6%	46%	low
memcached	**85**%	74%	13%	12%	low
belief	**206**%	80%	19%	10%	high
bfs	**190**%	24%	17%	12%	high
cc	**185**%	31%	17%	11%	high
pagerank	**183**%	23%	17%	11%	high
sssp	**193**%	10%	17%	11%	high
cassandra	**65**%	50%	14%	14%	low
mongodb	**130**%	95%	16%	14%	moderate

Table 4.1: Effect of the static NUMA policies in Linux on AMD48 with 48 threads

these applications, because it better balances the load and slightly improves the memory access locality. As a result, we have measured that the first-touch/Carrefour policy is only 2% slower in average than the best NUMA policy for these applications, with a worst case of 10% for `sp.C`.

To summarize this analysis, we confirm that all the studied NUMA policies are useful. This is explained by the fact that different sets of applications have different memory behaviors. More precisely, the round-4K/Carrefour policy is required for the "high" applications, the first-touch/Carrefour policy is required for the "moderate" applications, and the first-touch policy is required for the "low" applications.

This classification predicts, with a good accuracy, what is the best suited NUMA policy for a given application. However, classifying an application requires to measure this application behavior by executing it a first time. Moreover, the classification is relative to the hardware used for the metric measurement. This implies that the application must be run at least once with the first-touch policy to predict which policy to use for each hardware the application is deployed on. A good approach would be to provide an automatic classifier to the system administrator. The administrator is then able to classify each deployed application. Based on this classification, the system administrator chooses what policy to use when launching the application.

4.2 Current hypervisor NUMA management

In this section, we study the NUMA effects on virtualization. More precisely, we describe the Xen NUMA management policy, that we call round-1G, and explain the reason behind its design. We then explain what effects this policy has on the virtual machine memory layout and on the guest application memory access patterns. Finally, we evaluate how this Xen default NUMA policy affects the virtualized application performance.

The default round-1G policy

Xen uses a default NUMA policy, that we call the round-1G policy. When Xen creates a new virtual machine, it eagerly allocates the associated machine memory. Xen tries to pack the memory and the vCPUs of the new virtual machine on the minimal number of underloaded NUMA nodes by reserving a pCPU per vCPU. These NUMA nodes are called the *home-nodes*. If the system administrator explicitly pins a vCPU on a NUMA node, this NUMA node is included in the home-nodes. In our experiment, we always pin the vCPUs. As a consequence, we explicitly define the home-nodes of the virtual machines.

Xen favors locality by allocating the memory of a virtual machine from its home-nodes. It first tries, to allocate the memory by regions of 1 GiB with a round-robin algorithm from the home-nodes. In case of fragmentation or if the virtual machine needs less than 1 GiB (resp. 2 MiB), Xen allocates the memory by regions of 2 MiB (resp. 4 KiB). Because of the BIOS and I/O memory regions, the first and last GiBs of a virtual machine are always fragmented.

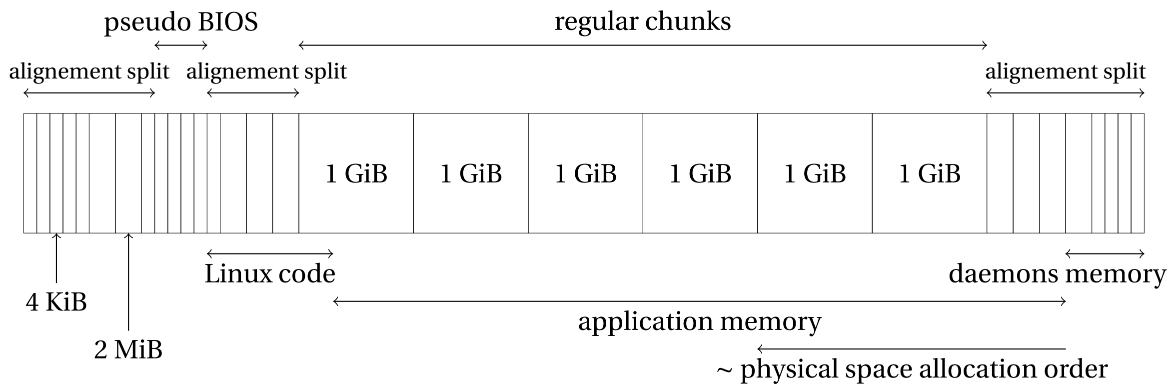

Figure 4.2: Typical physical space allocated with the default policy of Xen.

With this allocation strategy, the physical memory allocated for each virtual machine often follows the pattern illustrated in Figure 4.2. The first and last GiB of memory are fragmented in small regions of 4 KiB and some larger regions of 2 MiB, while the space in between is composed of 1 GiB regions evenly spread across the home nodes of the virtual machine. Because Xen allocates the regions with a round-robin algorithm, each consecutive region on Figure 4.2 is allocated on a different node.

Huge pages and splinterring

Allocating the virtual machine memory by regions of 1 GiB and 2 MiB instead of a page size is not a trivial choice. When the software accesses to the memory, the MMU translates the virtual address into a machine address. To this end, the MMU reads the page tables configured by the hypervisor and by the operating system. Reading a page table is a costly operation. To reduce the translation cost, the MMU uses a dedicated cache, the TLB. The TLB stores a limited amount of already translated page addresses.

Usually, the size of a memory page is 4 KiB. However, on modern processors, the operating system and the hypervisor can configure a page table to use larger pages of 2 MiB or 1 GiB, called *huge pages*. A TLB entry stores the translation of a page, whatever the size of the page. An operating system or a hypervisor usually maps large continuous memory regions with huge pages to reduce the number of used TLB entry. Using less TLB entries reduces the TLB miss rate and increases the application performance.

When the hypervisor enables the Hardware Assisted Paging, the TLB stores the translation from virtual address to machine address. The TLB does not store any intermediate translation to, or from, physical address. If the guest translation (from virtual address to physical address) and the host translation (from physical address to machine address) use different page sizes, a *page splinterring* occurs. In case of page splinterring, the smallest page size is used for the translation and the MMU adds an entry of this small page size in the TLB. Thus if the guest uses huge pages but the hypervisor only uses default 4 KiB pages, then the

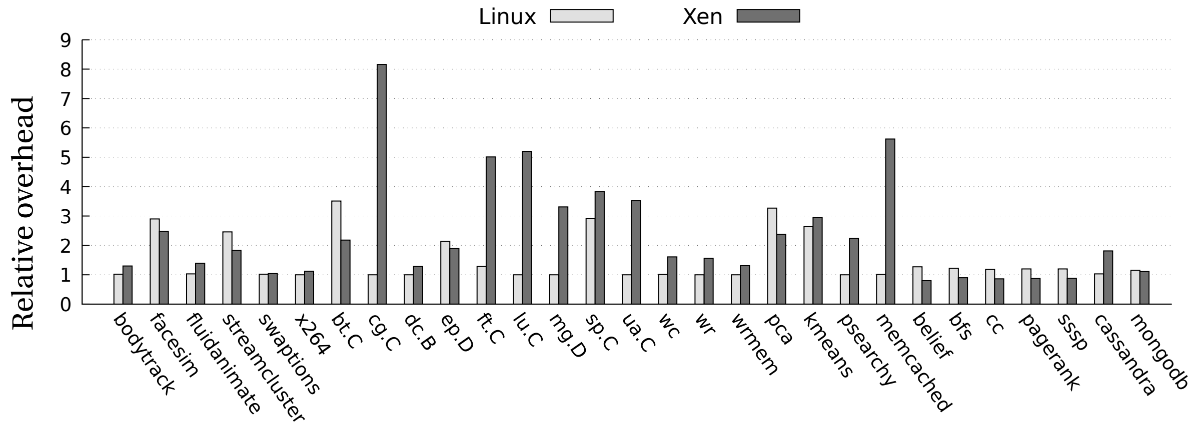

Figure 4.3: Relative overhead of Linux and Xen as compared to LinuxNUMA (lower is better).

MMU uses only 4 KiB entries: the host cancels the guest huge page optimization. For this reason, Xen tries to use 1 GiB huge pages. This allows the guest to use huge pages.

Evaluation of the default policy

In the following evaluation, we show the impact of the Xen default NUMA policy on the application performances. To this end, we execute the 29 applications of our benchmark in three configurations: LinuxNUMA is a native execution on Linux with the best NUMA policy we know for this application (see Section 4.1). Linux is a native execution on Linux with the default NUMA policy (i.e. first-touch). Xen is a virtualized execution on Linux/Xen with the default NUMA policy of Xen (i.e. round-1G). The Xen virtual machine has 48 vCPUs, each vCPU is pined to one pCPU. We configure the applications to use the 48 available CPUs. For these configurations, we enable the usage of `libactive` (see Section 3.2) for `facesim` and `streamcluster` and for the Xen configuration, we use the `pci-passthrough` driver (see Section 3.1) for `dc.B`, `bfs`, `cc`, `pagerank`, `sssp` and `mongodb`. We compare the overhead of Linux against LinuxNUMA and the overhead of Xen against LinuxNUMA. Since we remove the well-known bottlenecks, the overhead difference between the Linux and the Xen configurations is probably a consequence of their NUMA management (we confirm this hypothesis in Chapter 5).

The Figure 4.3 shows the relative overhead of each application with the configurations Linux and Xen compared to LinuxNUMA. The first observation we can make is that the Xen configuration has a negative impact on application performance: 17 of the 29 applications are slowed down by more than 50%, 12 of them by more than 100% and this slowdown reaches 700% for `cg.C`. Since we removed the already known bottlenecks (see Chapter 3), we highly suspect that this negative impact comes from the default NUMA policy of Xen.

Additionally, we observe that the suspected impact of the NUMA policy is higher on Xen than on Linux: only 7 applications on Linux are slowed down by more than 50% with a

maximum of 250% for `bt.C`. Moreover, the applications slowed down on Linux are not the ones slowed down on Xen. As we saw on Section 4.1, the impact of the NUMA policy on an application largely depends on the memory access pattern of this application and different NUMA policies bring better performance applied to different memory access patterns.

As explained above in Section 4.2, the default NUMA policy of Xen is to allocate chunks of 1 GiB in round-robin over the NUMA nodes. While this allocation pattern enables the usage of huge pages, it has a significant impact on both the access locality and on the load imbalance. On the first side the applications have a poor locality. Linux allocates the application memory without any knowledge of the NUMA topology. As a consequence, the allocated memory is unlikely on the same NUMA node than the accessor thread. On the other side, the memory load is imbalanced. Indeed, the Linux guest sees a UMA hardware and tends to allocate memory in continuous regions. As a result, the allocated memory is likely to be on a unique 1 GiB region on a single NUMA node for applications with small memory footprints. This explains why the default Xen NUMA policy has such a negative impact on application performances, and especially on applications of the *low* group (see Table 4.1) since they nonetheless experiment bad data locality that we know is an important factor, but also bad memory balance. We can confirm this by observing that the 7 applications with an overhead on Xen higher than 200% are `cg.C`, `memcached`, `lu.C`, `ft.C`, `sp.C`, `ua.C` and `mg.D`, which are applications of the *low* and *moderate* groups.

4.3 Conclusion

This Chapter shows the impact of NUMA memory management strategies on applications performance. In a first part, we show that, on a 48 cores NUMA hardware and a large set of applications, the choice of an appropriate NUMA memory management policy on a native configuration improve the performance up to 250%. We also confirm that, in some cases, Carrefour [25] improves the applications performance as compared with standard Linux policies described in Section 2.2. In this first part, we finally propose a metric to choose what is the best policy for a given application on a given hardware. In a second part, we describe the default NUMA management policy of the Xen hypervisor. Along with this description, we detail what effects of this policy on the virtual machine memory layout. We then show, with an evaluation of this policy compared to native execution times, that the virtualization causes an important performance overhead, up to 700%. We explain that is overhead is probably a consequence of the virtual machine memory layout, caused by the Xen NUMA policy.

CHAPTER 5

VIRTUALIZATION OF NUMA ARCHITECTURES

In Chapter 4, we show the impact of NUMA policies on the native application performance. Each application has a specific memory access pattern. Each NUMA policy better fits with a specific access pattern. Choosing an appropriate NUMA policy for a given application is thus essential to bring good performance. We also investigate the effects of virtualization on a NUMA hardware. We show that, in a virtualized configuration where we remove the well-known virtualization bottlenecks (see Chapter 3), using a large number of CPU on a NUMA hardware leads to a huge performance overhead compared to a native execution. We suspect that the NUMA policy of the Xen hypervisor is the cause of this overhead.

In this chapter, we show how to bring good application performance on a virtualized NUMA hardware. We assume that the overhead we observe in Section 4.2 between native and virtualized configuration comes from the Xen round-1G NUMA policy. We also know that one of the native policies, first-touch, round-4K or Carrefour, enables good application performance. Our goal is thus to use these native policies in virtualized configurations. Using these policies directly in the guest system is not possible. Indeed, as we explain in Section 2.3, the virtualization isolates the guest system from the hardware. The Linux guest does not see the NUMA topology and cannot use NUMA policies. We see two methods to solve this problem: (i) modify the hypervisor to expose the NUMA topology to the guest or (ii) implement NUMA policies in the hypervisor, which has access to the NUMA topology.

In this thesis, we choose to implement NUMA policies in the hypervisor. We start this chapter by explaining why. Then, we explain how we implement the NUMA policies in the Xen hypervisor. Finally, we evaluate these policies. In this evaluation, we show that we improve the virtualized application performance up to 500% compared to the round-1G policy. Additionally, we show that using an efficient NUMA policy reduces the virtualization overhead

below 50% for most of the applications. This last observation confirms that the round-1G NUMA policy is the cause of the overhead observed in Section 4.2.

5.1 Improved hypervisor NUMA management

In this chapter, we provide a way to use the native policies, first-touch, round-4K and Carrefour, in virtualized configurations. One could be tempted to delegate this task to the guest operating system, as it already implements these policies. On a classical hypervisor this is not possible: the hypervisor hides the actual NUMA topology to the guest, showing instead a flat UMA topology. With no information about how the vCPUs are clustered on the NUMA nodes, and how the memory is spread across the NUMA nodes, the guest operating system cannot take any useful NUMA related decision.

Some hypervisors, like the Amazon EC2 modified Xen hypervisor, solve this problem by exposing the NUMA topology to their guests. However, we see many drawbacks to this approach. First, exposing the hardware creates a security issue. A malicious guest could use the hardware information to select specific attacks exploiting known flaws of this hardware. Second, exposing the NUMA topology may prevent virtual machine optimizations. Several works [66, 70] migrate the vCPUs between the pCPUs to balance the load. Such a migration would result, from the guest point of view, in a physical topology modification. As much as we know, none of the mainstream operating systems is able to handle such online topology modifications. Third, a knowledge of the NUMA topology is necessary but not sufficient to take efficient NUMA placement decisions. A knowledge of the resource usage is also necessary to prevent contention. Typically, two virtual machines that see the NUMA topology could allocate hot data on the same node. For each of the guest systems, this would be an efficient decision. However, from a global point of view, the total load on the allocation NUMA node could saturate the memory controller, inducing a performance degradation for the two virtual machines.

Instead of exposing the NUMA topology to the guest, we choose to implement the NUMA policies in the hypervisor. We consider that the hypervisor uses a single NUMA policy for a whole virtual machine. Indeed, in the cloud computing, clients often use one virtual machine per task. A single virtual machine thus executes a unique application in addition to system daemons. Additionally, as we state in Section 4.1, the NUMA policy to use varies from an application to another. For this reason, the hypervisor should allow the guest system administrator to choose what NUMA policy to use.

In the following sections, we explain how we implement each of the native NUMA policies in the Xen hypervisor. For the first-touch and Carrefour policies, we encounter limitations. We describe these limitations, how they affect our implementation and we explain how future works could overcome them. We also present an interface allowing the guest administrator to choose the NUMA policy to use from inside the virtual machine.

The Xen implementation of round-4K

Just like the Xen default round-1G policy, the round-4K policy allocates memory chunks in round-robin among the possible NUMA nodes while there is available memory on them. Unless the Xen default policy which allocates chunks with the largest possible size, the round-4K policy systematically allocates chunks as small as possible. As explained in Section 2.2, this difference ensures that, under the round-4K policy, application memory is as spread as it could be among the NUMA nodes of the system, and thus, the memory load is balanced among the memory controllers.

Implementing the round-4K policy in Xen is straightforward. We reuse the round-1G allocation algorithm and disable allocation of memory regions larger than 4 KiB. Since Xen allocates its memory eagerly, at the virtual machine boot, the guest administrator must choose to use this policy before to boot the virtual machine. This is not a problem in practice since, as we show in Section 5.2, the round-4K policy outperforms the round-1G policy for all the applications we evaluate. We thus set the round-4K allocation policy as the new default policy of Xen. Also, there is no technical limitation that prevent to switch from a round-4K memory layout to a round-1G memory layout after the virtual machine boot. Indeed, the hypervisor is able to transparently migrate pages from a machine address to another one.

The Xen implementation of Carrefour

Section 2.2 describes the principles of the Carrefour policy. In this section, we briefly remind these principles. We also describe the native implementation and we explain how we adapt this implementation to the Xen hypervisor.

Carrefour is a dynamic policy that detects memory load imbalance or bad access locality and migrates memory pages while the application is running. To this end, Carrefour uses two sets of counters: the Performance Monitoring Counters (PMC) and the sampling counters (IBS[1]). The PMC are cheap to use and Carrefour uses them to decide if the application intensively uses the memory. If this is not the case, Carrefour remains idle. If the application uses the memory, Carrefour uses the IBS counters to identify the hot pages of the application and what cores access them. Based on this information, Carrefour either (i) dispatches the hot pages on the available NUMA nodes to balance the memory load, (ii) migrates the pages on the same NUMA node than the accessor cores if they are all on the same node, (iii) replicates the pages accessed in read mostly mode on the NUMA nodes.

Figure 5.1 illustrates the software architecture of Carrefour in a native configuration. Carrefour is composed of two distinct parts: the userland component and the privileged component. The userland component uses the PMC to make a coarse measure of the machine state. A PMC is a pair of hardware registers: a control register and a count register. Carrefour configures the type of event to count (accesses to the memory, see Section 2.2) by writing

[1] IBS stands for Instruction Based Sampling and is the AMD version of sampling counters. Intel provides Precise Event Based Sampling counters. We explain in Section 5.1 why we only discuss AMD counters.

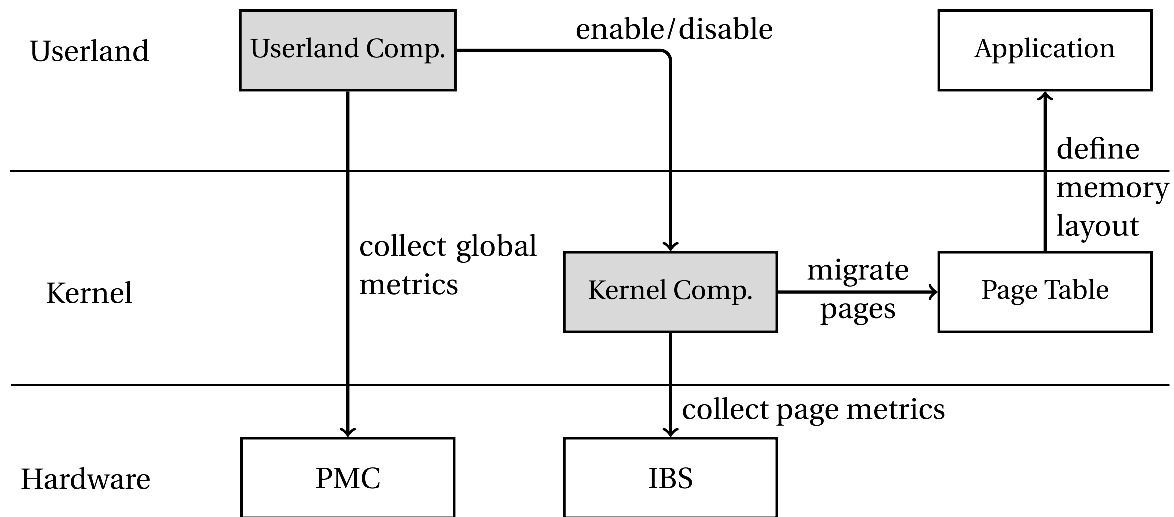

Figure 5.1: Architecture of the native version of Carrefour. The modules filled with gray are the Carrefour components.

on the control register. Each time the event occurs, the count register is incremented. The two registers are only available from kernel mode. The userland module of Carrefour uses a system call to access them. In our Xen implementation, the userland module executes in the dom0 (i.e. the privileged administration guest) and accesses the PMC through a new hypercall we add to Xen.

Using correctly the PMC is not trivial. There is only a limited amount of PMC, typically there are only four PMC per core on our experimental machine. Moreover, some events cannot be counted simultaneously by more than one core. This is typically the case for the amount of access to shared hardware components. Finally, some events are discarded when the counting core is halted[2]. In a native configuration, the operating system handles these aspects and provides a simple abstract view of the counters to the userland Carrefour component. Xen has no such mechanism. We thus choose to provide raw access to them through hypercalls. We modify the Carrefour userland component to carefully use the performance counters assuming an exclusive control on the PMC.

The second component of Carrefour executes in the most privileged mode of the machine: in kernel mode for native configurations and in host mode for virtualized configurations. This second component uses the IBS counters and performs the page migrations. There is one IBS counter per core, composed of several control registers, one countdown register and several information registers. Carrefour configures the type of event to sample (any write on main memory) by writing the control registers. Carrefour also configures the sampling frequency by setting an initial value to the countdown register. Each time the sampled event occurs, the countdown registers is decremented. When the countdown reg-

[2]When the only load of the machine is a single application with synchronized threads, the cores go to halt mode quite often.

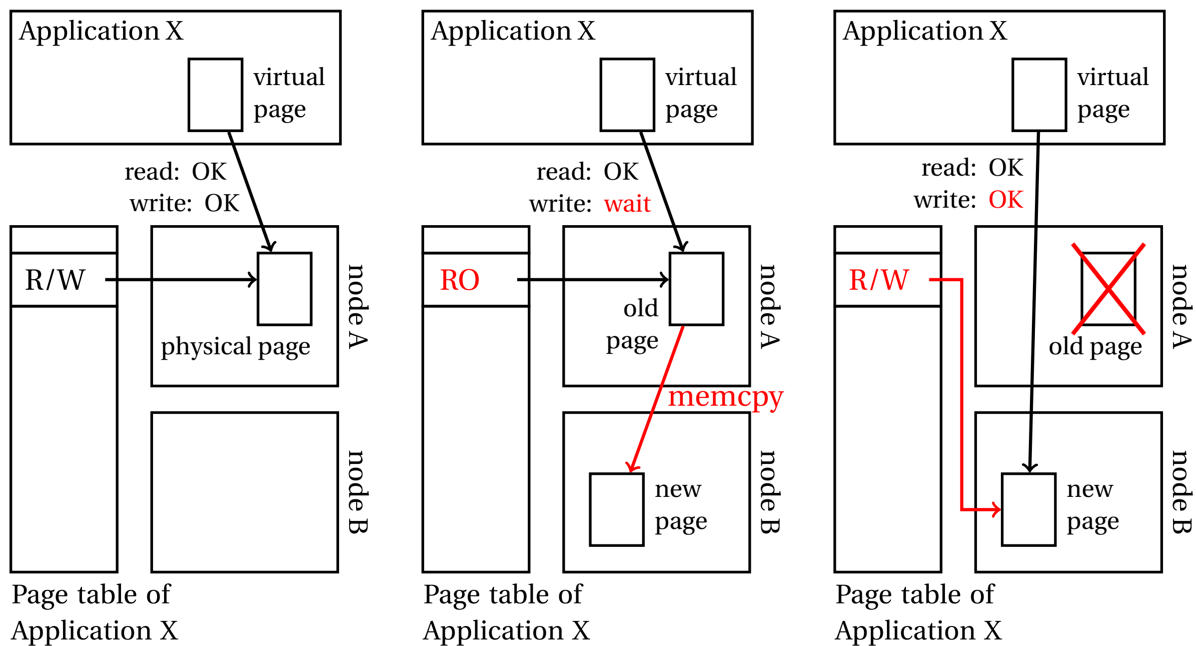

Figure 5.2: Transparent page migration. The leftmost picture is the initial state with the virtual page pointing on the source node (node A). The middle picture is the transitory state with the virtual page pointing in read-only on the old page while the system copies the page to the new node. The rightmost picture is the final state where with virtual page pointing on the target node (node B).

ister reaches zero, the processor fills the information registers. In our case, the information registers contain the virtual addresses of the last memory access. The processor also triggers an exception, handled by the system. On the native Carrefour, Linux creates a tuple containing (i) the sampled virtual address, (ii) the current process id and (iii) the id of the current core, which is the accessing core. The tuple is then stored for future use.

The privileged module of Carrefour periodically computes the set of hot pages: the pages with more than two associated tuples. The tuples also indicate which core access the pages. Based on this information, Carrefour decides which page to migrate or replicate as we explain in Section 2.2. To migrate a page, the native version of Carrefour removes the write permission on the page. Carrefour then copies the page on the target NUMA node. If the application attempts to write on the page during the copy, the processor triggers a page fault and the application blocks. When the copy is done, Carrefour updates the page table so the virtual address of the page is now associated to the new physical address, on the target NUMA node. Carrefour also gives back the normal permission to the page, so the blocked application threads continue. Figure 5.2 illustrates the whole page migration process. We can notice that Carrefour needs the process id and the virtual address of the page to modify it in the page table.

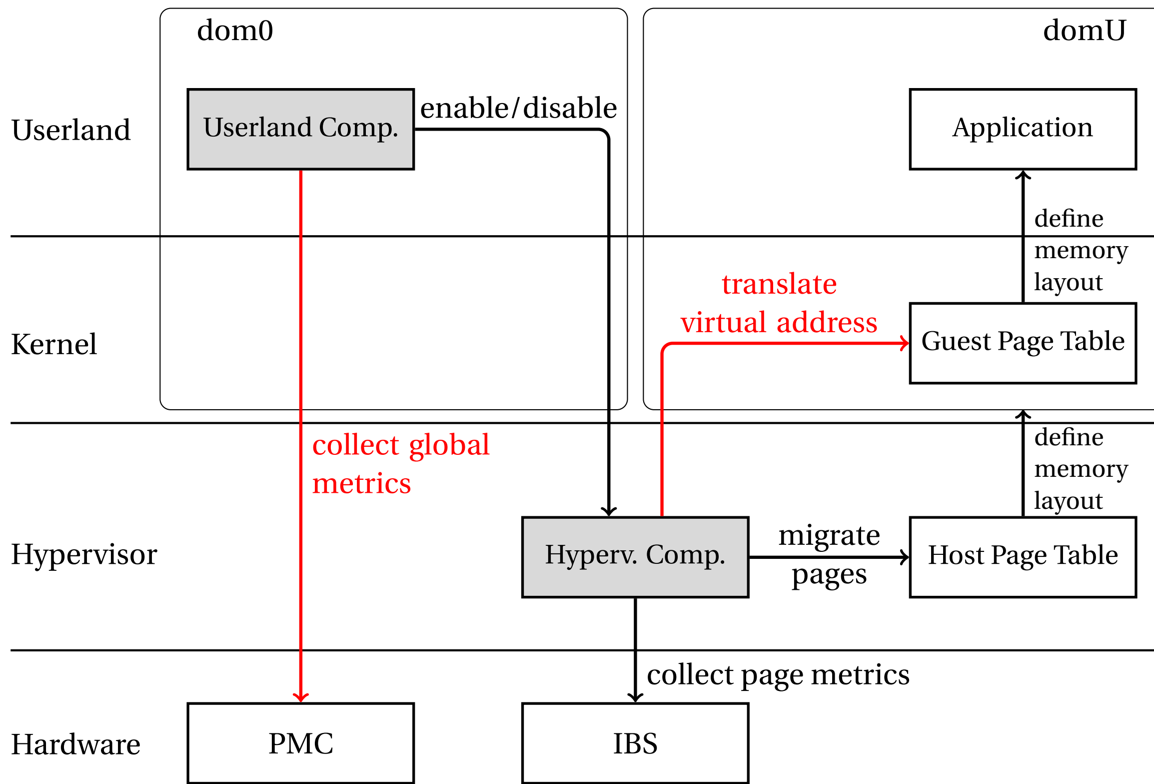

Figure 5.3: Architecture of the virtualized version of Carrefour. The modules filled with gray are the Carrefour components. The links in red are the modified or added links compared to the native version.

On the virtualized Carrefour, an additional step is required during sampling. When the processor triggers a sampling exception, the hypervisor sees the current virtual machine id instead of the current process id. Also, the hypervisor still sees the sampled virtual address. However, a virtual machine id and a virtual address are not sufficient to identify a page: several processes of the same virtual machine may use the same virtual address. Additionally, the hypervisor migrates pages by updating its own page tables (see Section 2.3). The hypervisor needs the virtual machine id and the physical address of the page to modify the page tables. For this reason, the virtualized Carrefour must translate the sampled virtual address into its physical address. To this end, Carrefour has to walk the guest page table. We are sure that the guest system respect the page table structure because it is intended to be read by the hardware. However, in a multicore context, nothing prevents the guest to modify the page table while the hypervisor is walking it.

A safe approach to translate virtual addresses would be to stop the entire virtual machine during the walk, or to write protect the page table as the hypervisor reads it. These approaches are slow and difficult to implement. Instead, we observe that the Carrefour privileged component relies on statistics. Indeed, nothing ensures that a page having a lot of

associated tuple is effectively a hot page. Carrefour may then compute a spurious vision of memory accesses, leading to bad decisions. However, with a good probability, the Carrefour vision of memory accesses is valid when the amount of sampled access is high. In the same way, a virtual address translation may fail because of concurrent modification by the guest. With a good probability, however, Carrefour successfully translates virtual addresses to physical addresses without synchronizing with the guest. Figure 5.3 illustrates the software architecture of Carrefour in a virtualized configuration.

Limitations of PEBS with virtualization

In Section 5.1, we only discuss the IBS counters, which is the AMD implementation of sampling counters. Intel provides its own implementation, named PEBS. The native version of Carrefour can use PEBS, however, the virtualized version of Carrefour cannot. Actually, the PEBS counters are unusable when the Hardware Assisted Paging is enabled. The reason is that PEBS stores the sampled information in memory instead of dedicated registers. The hypervisor must configure the PEBS storage address by writing its virtual address in a dedicated register. However, when the Hardware Assisted Paging is enabled, the hypervisor has no control over the mapping of virtual addresses. Instead, the guest system chooses to allocate virtual space to itself or to its applications. If the processor samples an event while the guest is already using the PEBS virtual address, then the processor overwrite the guest data. In the best case, this causes a guest system crash. In the worst case, this silently corrupts a guest application memory.

Several methods could be used to circumvent this problem. The first one would be to write protect the part of the guest page tables that maps the virtual address of the PEBS buffer. The hypervisor could then intercept the guest allocation of the PEBS virtual address and reconfigure the PEBS to use another virtual address. Another way would be to use paravirtualization. The hypervisor would reserve virtual addresses in the guest system and use these addresses for the PEBS storage area. As these two techniques would require complex engineering, we choose to not implement them and to focus on the AMD hardware.

The Xen implementation of first-touch

As we explain in Section 2.2, the first-touch policy provides a good access locality as long as each application thread uses mostly the memory zones it allocates. As shown in Figure 4.1, many applications reach good performance with this allocation policy. In a native configuration, Linux allocates the physical memory lazily. When the application asks for memory, Linux allocates a virtual address range but does not map it on any physical memory. Later, when the application accesses these virtual addresses, the MMU triggers a page fault. Only then, Linux allocates physical memory from the NUMA node performing the access and maps the faulting virtual address on it.

This first-touch implementation is not possible for a hypervisor. On one side, it is a usual operation for an application to ask and release memory. On the other side, there is no concept of “asking memory” or “releasing memory” for a guest operating system. Indeed,

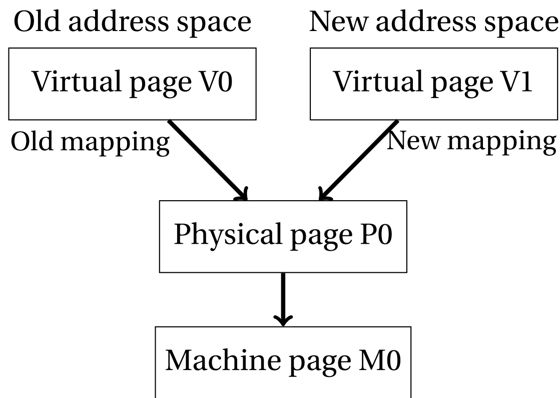

Figure 5.4: Issue to implement first-touch in Xen.

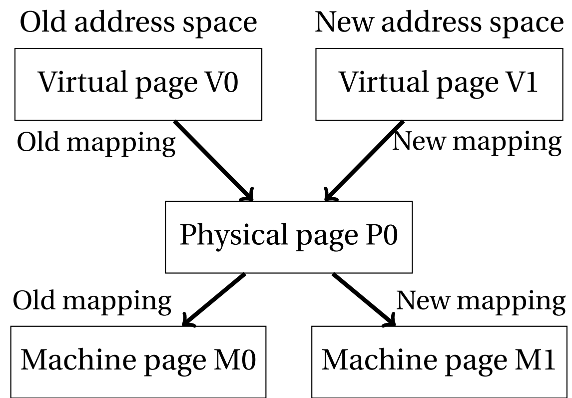

Figure 5.5: Implementation of first-touch in Xen.

an operating system is designed to make the best usage of a fixed amount of hardware resources. For this reason, the hypervisor cannot simply allocate the guest physical memory lazily. The guest operating system uses the same physical addresses for many different purpose during the virtual machine lifetime: boot time scripts, I/O cache, main application, etc... Figure 5.4 illustrates this issue. Beside this, it is not possible for the hypervisor to detect application level memory allocation. Indeed, with Hardware Assisted Paging, the processor delivers the guest application page fault directly to the guest system without involving the hypervisor.

We choose to implement first-touch in the Xen hypervisor with paravirtualization. We introduce a new hypercall usable from the guest kernel mode. This hypercall is used by the guest operating system to indicate that a physical page is free (i.e. it does not contain any useful data anymore). This hypercall has the same semantic than the Linux syscall `madvise(..., MADV_FREE)`. From the guest point of view, invoking this hypercall on a physical page is the same as filling the page with zeroes. From the hypervisor point of view, this hypercall means that the physical page can be unmapped and the corresponding machine page can be released. If the guest operating system uses this unmapped physical page later, the MMU delivers a *nested page fault* to the hypervisor. The hypervisor can thus allocate a machine page from the appropriate NUMA node, like for the native first-touch policy. Figure 5.5 illustrates this solution.

With the physical release hypercall, the Xen hypervisor effectively implements a first-touch policy. This implementation however brings two issues. First, the first-touch policy leads to virtual I/O failures when using the IOMMU. This first issue is specific to Xen and discussed in Section 5.1. Second, releasing the physical pages is an expensive operation. Indeed, a hypercall is far more expensive than a system call. The x86 system call instructions are optimized to be low latency by minimizing the amount of operation and security check the processor performs. The processor only does a few memory read and some operations on registers. To the contrary, during a hypercall, the processor stores the state of the

virtual registers in memory and restore the state of the host registers from memory (see Section 2.3). This load and save roughly costs an entire 4 KiB memory read and write. As a result, a naive implementation of the first-touch policy with one hypercall by page release would result in a dramatic performance drop.

We address the hypercall cost issue by slightly modifying the hypercall. The physical release hypercall takes a queue of physical page addresses to release instead of a single one. This technique amortizes the cost of an hypercall proportionally to the size of the queue. On the guest side, all the vCPUs share the same queue of released pages. The queue is then protected by a lock. Batching the page releases however raises a new challenge. The operating system may reallocate a page while it is in the page queue but not yet sent to the hypervisor. When the hypervisor receives the page queue, it can thus not ignore the content of a page as the page may already be reused by a process. Without any other mechanism, the hypervisor would thus have to copy the old content of the page when it migrates the page to a new NUMA node, which is expensive.

We solve the problem by trapping both the page allocation and release of the guest operating system. Each entry in the queue contains a pair `(op, page)`, in which `op` is the operation (allocation or release) and `page` is the address of the physical page. When the guest operating system allocates or releases a page, it acquires a lock before adding the pair to the queue. Then, before releasing the lock, the guest operating system sends the queue to the hypervisor through an hypercall when the queue is full. The guest operating system has to keep the lock during the hypercall in order to ensure that another core cannot reallocate a free page of the queue during the hypercall.

When the hypervisor receives the queue, it starts with the most recent operations, i.e., the end of the queue. Then, the NUMA policy keeps a list of the visited pages and only takes into account the most recent operation associated to a page. If the most recent operation is a release, the hypervisor knows that the physical page is no longer used and it can invalidate its entry. If the most recent operation is an allocation, the hypervisor knows that the page may already be reused by a process. This case is rare and we simply handle it by letting a reallocated page on its current node. Indeed, copying the old content of a page would be too expensive in the common case and would thus not be efficient.

Finally, using a single global queue protected by a lock is a bottleneck when the virtual machine uses many cores. As a final solution, we partition the global queue in independent queues. We associate each page address to a single queue by using the two less significant bits of the page frame number. As a result, each queue has its own independent lock, which increases the parallelism. With the partitioned queue, we have measured that 87.5% of the time is spent invalidating the pages during an hypercall, while sending the queue only takes 12.5% of this time. For this reason, we have not used more efficient and scalable queue algorithms [37].

Limitations of first-touch with IOMMU

The previous section indicates the impossibility to use the first-touch strategy while using the IOMMU. This is actually a design incompatibility more than a conceptual issue. As described in Section 3.1, the IOMMU is the device responsible for translating physical addresses to machine addresses for I/O devices. This translation allows a guest to have full access to a device without the need to intercept the requests with the hypervisor and thus enables virtualized I/O with no overhead.

Besides it, the first-touch policy relies on lazy allocation. The hypervisor leaves physical addresses unmapped. When the processor accesses an unmapped physical address, the MMU triggers an exception and expect the hypervisor to handle the cause of this fault. The hypervisor then allocates machine memory, updates the page table and gives control back to the main process (see Section 5.1). Because the IOMMU is connected to the faulting core by an external bus, the IOMMU cannot act synchronously with the core. Thus, when an I/O device tries to access to an unmapped physical address, the IOMMU notifies the guest system that a hardware error occurred. The IOMMU does not notify the hypervisor.

The hypervisor impossibility to handle IOMMU faults prevents it from using the first-touch policy along with the Xen pci-passthrough driver. Several methods could be used to circumvent this issue. The most straightforward approach would be for the guest to systematically touch any page (i.e. reading its first byte) before to require a DMA access on it. This however may need modifications in several places in the code of the guest including in the many driver codes. A more transparent approach would be to affect different permissions to MMU and IOMMU on the same physical page. As described in Figure 3.1 and Figure 3.2, different bits are used for permissions of the processor and I/O peripherals. It is then possible to let the released page mapped but forbidden to the guest by clearing the `U` bit[3] but setting the `IR` and `IW` bits to let the DMA peripherals work as usual.

The Xen NUMA policy selection interface

In the previous sections, we explain how to implement NUMA policies in a virtualized configuration to bring good applications performance. This is not sufficient, as stated in Section 5.1, the virtual machine administrator should be able to choose what policy to use depending on the application to execute, helped by the policy selection metric described in Section 4.1.

To this end, the modified version on Xen provides a new hypercall to select what policy to use. As stated in Section 5.1, the round-4K policy is the new default policy, replacing the round-1G policy. A new virtual machine thus always boots its memory interleaved with a 4 KiB granularity.

The guest administrator can enable Carrefour and first-touch while the virtual machine is running. Just like its native version, Carrefour is a dynamic policy and thus can be enabled or disabled at any time. The only requirement is to have the Carrefour userland component

[3] A guest can only access physical pages with the *user* permission.

always running in the *dom0*, ready to be used. This Carrefour component can be shared between several guests wanting to enable the Carrefour policy for their own benefits. Enabling the first-touch policy means to accept the release hypercalls described in Section 5.1. When disabled, the hypervisor only zeroes the page to respect the specification of the hypercall from the guest point of view while letting them mapped. When using the first-touch policy, the guest system administrator should configure the operating system to limit the memory usable by the I/O cache. Indeed, if the guest operating system uses all the free pages for I/O cache instead of releasing them, the first-touch policy cannot work.

5.2 Evaluation

In this section, we evaluate the NUMA policies implemented in Xen to answer several questions. Does the impact of the NUMA placement is as high in a virtualized configuration than in a native one ? Does an appropriate NUMA policy used in a virtualized configuration reduce the execution overhead of applications on large NUMA machines ? Does the metric presented in Section 4.1 is still valid in a virtualized configuration ? To this end, we evaluate the execution time of the 29 applications of our benchmark suite, described in Section 2.4, under the implemented NUMA policies, described in Section 5.1, using the hardware configuration described in Section 2.2.

Evaluation of vNUMA on a single machine

We describe here some experiments with a single native application running, aside from mostly idle system daemons. Each application uses as many threads as the number of core or vCPU. For virtualized configuration, we execute a single domu, consolidated with the dom0. The dom0 only executes the Carrefour userland component and the benchmarking scripts that remain idle most of the time. The domu uses as many vCPUs as the number of pCPU. We use the `tc` pining strategy in native configurations and the `vctc` pining strategy in virtualized configurations (see Section 3.3). For these configurations, we enable the usage of `libactive` (see Section 3.2) for `facesim` and `streamcluster` and for the Xen configuration, we use the `pci-passthrough` driver (see Section 3.1) for `dc.B`, `bfs`, `cc`, `pagerank`, `sssp` and `mongodb`.

Comparison of the virtualized NUMA policies

In a first experiment we execute the 29 applications of our benchmark suite under four different NUMA policies: round-4K, round-4K/Carrefour, first-touch and first-touch/Carrefour. For each application, we compare the execution times under these policies to the execution time under the default round-1G policy. With this comparison, we want to evaluate the impact of the NUMA policy on the virtualized application performance.

Figure 5.6 reports, for each of the 29 applications, the speedup for each NUMA policy as compared to the round-1G policy. First, we observe that using an efficient NUMA policy drastically improves the performance. Indeed, for 9 applications, the execution time is

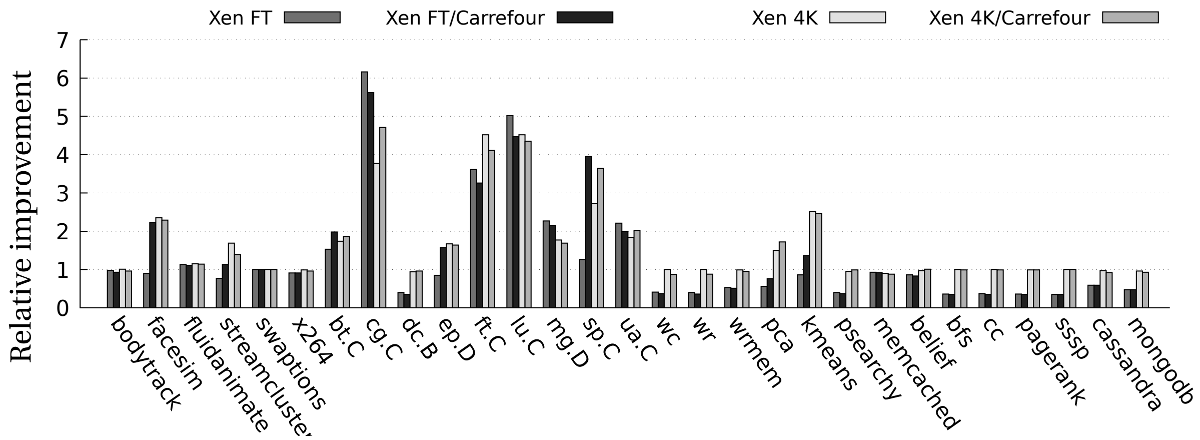

Figure 5.6: Relative improvement of the Xen NUMA policies as compared to the default round-1G policy (higher is better).

improved by more than 100%. In the best case, for `cg.C`, the completion time is divided by 6. Moreover, we observe that each NUMA policy yields the best performance for some applications. The first-touch/Carrefour policy is the best for `bt.C` and it improves the performance by 100%, the round-4K/Carrefour policy is the best for `sp.C` and it improves the performance by 290%, the first-touch policy is the best for `kmeans` and it improves the performance by 170%, and the round-4K policy is the best for `ft.C` and it improves the performance by 315%.

Additionally, we observe that the default round-1G policy is much less efficient than the NUMA policies we implement. The round-1G policy is only better than the other policies for four applications. For these applications, as presented in Figure 5.6, if we replace the round-1G policy by the second best policy, the maximum performance degradation we observe is 10%.

Finally, we observe that for the disk intensive applications improved by the pci-passthrough driver (`dc.B`, `bfs`, `cc`, `pagerank`, `sssp` and `mongodb`, see Section 3.1), the first-touch policy seems to systematically degrade their performance as compared to the round-1G policy. For these applications, we disable the pci-passthrough driver when we enable the first-touch policy (see Section 5.1). This is the cause of the observed overhead.

Comparison between native NUMA and virtualized NUMA policies

In a second experiment, we execute the 29 applications of our benchmark suite in three configurations. The first configuration is LinuxNUMA, a native configuration with the best known NUMA policy for each application. The second configuration is Xen, a virtualized configuration with the default round-1G NUMA policy. The third configuration is Xen-NUMA, a virtualized configuration with the best known NUMA policy for each application. We indicate the best NUMA policies for native and virtualized configurations in Table 5.2. The other parameters (pining, amount of threads, etc. . .) are the same as in the previous

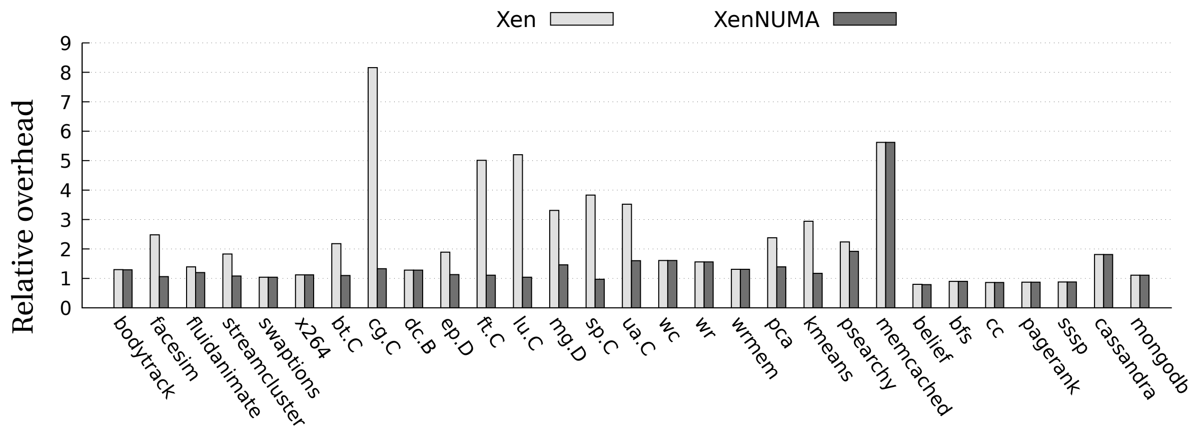

Figure 5.7: Relative overhead of Xen and XenNUMA as compared to LinuxNUMA (lower is better).

experiment. With this experiment we show that the large overhead of Xen is caused by an inefficient NUMA placement and not by other virtualization overheads. To this end, we compare the overhead of Xen against LinuxNUMA and the overhead of XenNUMA against LinuxNUMA. The only difference between the Xen and XenNUMA configurations is the used NUMA policy. Consequently, any overhead difference between these configurations is caused by the NUMA effects.

Figure 5.7 reports the overhead of Xen and XenNUMA as compared to LinuxNUMA. We observe that with efficient NUMA policies, only 4 applications remain degraded by more than 50%, while 14 applications have an overhead of more than 50% with Xen. This result shows that a large overhead of virtualization on large multicores is caused by the NUMA placement policy. We have shown that by integrating inside Xen the efficient NUMA policies that have been implemented within the context of operating systems, we successfully reduce the virtualization overhead.

For the 4 remaining applications, we can observe that `memcached`, `cassandra`, and `ua.C` frequently leave the CPU (column context switch of Table 2.8). These applications suffer from the cost of virtualized IPIs. They are not corrected by our `libactive` solution, which targets the pthread locks and condition variables. `Memcached` and `cassandra` continuously wait for network packets because they intensively use the network. `Ua.C` intensively uses an ad-hoc synchronization mechanism that relies on the Futex of Linux. These applications are thus probably slowed down by the cost of virtualized IPIs. `Psearchy` intensively uses the hard drive (see Table 2.8) and, despite the use of the IOMMU, may activate a bottleneck in the I/O stack that we have not yet identified.

Selection of native and virtualized NUMA policies

In this experiment, we compare the best policies in both native and virtualized configuration. The goal of this comparison is to know if the best NUMA policy for an application is

	LinuxNUMA	XenNUMA
bodytrack	Round-4K	Round-4K
facesim	Round-4K	Round-4K
fluidanimate	Round 4K / Carrefour	Round 4K / Carrefour
streamcluster	Round-4K	Round-4K
swaptions	Round-4K	Round-4K
x264	First-Touch	Round-4K
bt.C	First-Touch / Carrefour	First-Touch / Carrefour
cg.C	First-Touch	First-Touch
dc.B	First-Touch	Round-1G
ep.D	Round-4K	Round-4K
ft.C	Round-4K	Round-4K
lu.C	First-Touch	First-Touch
mg.D	First-Touch	First-Touch
sp.C	Round 4K / Carrefour	Round 4K / Carrefour
ua.C	First-Touch	First-Touch
wc	First-Touch / Carrefour	Round-4K
wr	First-Touch	Round-4K
wrmem	First-Touch	Round-4K
pca	Round-4K	Round 4K / Carrefour
kmeans	Round-4K	Round-4K
psearchy	First-Touch	Round-4K
memcached	Round 4K / Carrefour	Round 4K / Carrefour
belief	Round-4K	Round 4K / Carrefour
bfs	Round-4K	Round-4K
cc	Round 4K / Carrefour	Round 4K / Carrefour
pagerank	Round 4K / Carrefour	Round 4K / Carrefour
sssp	Round 4K / Carrefour	Round 4K / Carrefour
cassandra	First-Touch / Carrefour	Round-1G
mongodb	First-Touch / Carrefour	Round-1G

Table 5.1: Best NUMA policies.

the same in native and virtualized configurations. We know that the metric described in Section 4.1 indicates, with a good accuracy, the best native NUMA policy. If the best native and virtualized NUMA policies are the same, then our metric is also efficient to predict what policy to use inside a virtual machine.

Table 5.1 shows what are the NUMA policies used for each application in both LinuxNUMA and XenNUMA. We observe that, among the 29 executed applications, 19 of them execute the fastest with the same NUMA policy. In the applications not having the same best NUMA policy, 3 of them, `wc`, `wr` and `wrmem` make memory allocations at a high rate, thus stressing the first-touch hypercall. The overhead of this hypercall is sufficient to make the best policy become round-4K instead of first-touch or first-touch/Carrefour.

For 2 other applications, `pca` and `belief`, the best policy only differs by an enabled Carrefour in the virtualized configuration along with a round-4K static policy. As we saw in Section 2.2, Carrefour is useful either to spread contention prone memory areas or to relocate pages on an exclusive accessing node. As the round-4K policy already perfectly spread the memory, we can assume only the second mechanism (the migration heuristic) is used in round-4K/Carrefour. An explanation could be that some structures of the kernel could be relocated to reach better performances, typically the *per CPU* structures, since, in a virtualized configuration, the kernel memory can also be migrated by Carrefour.

The 5 remaining applications, `x264`, `dc.B`, `psearchy`, `mongodb` and `cassandra` have com-

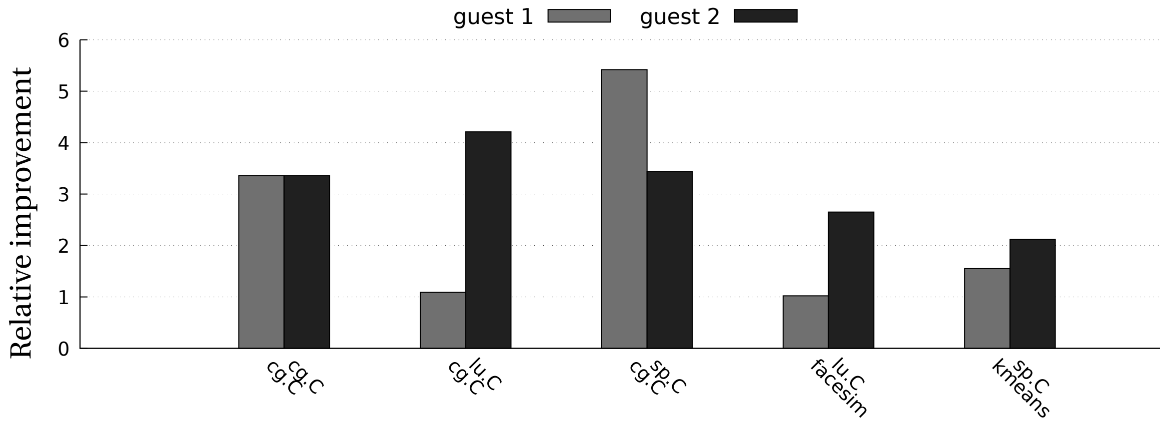

Figure 5.8: Relative improvement of XenNUMA over Xen with 2 collocated VMs (24 cores each, higher is better)

pletely different best NUMA policies in native and virtualized configuration. For `dc.B`, `cassandra` and `mongodb`, the best virtualized policy is round-1G, which is not available in a native configuration. The 2 remaining applications, `x264` and `psearchy` are the fastest with first-touch in LinuxNUMA and round-4K in XenNUMA but we have no definitive explanation for them.

Evaluation of vNUMA on several machines

Evaluating the execution time of applications in single virtual machines is convenient to exhibit the NUMA policy impact on performances. However, this is not a realistic scenario. In this section, we present an evaluation of the virtualized NUMA policies in workloads with two virtual machines. In the following experiments, each virtual machine executes a single application with as many threads as available vCPUs in the virtual machine. For each of the applications, we select the best Xen NUMA policy (see column `XenNUMA` of Table 5.1). The figures report the improvement of the best NUMA policy over the default NUMA policy on Xen.

Collocated virtual machines

In Figure 5.8, each virtual machine has 24 vCPUs. We pin the first virtual machine on one half of the NUMA nodes and the second virtual machine on the other half. Each physical CPU thus executes a single vCPU. We have observed that, for some of the applications, performance varies when we select different NUMA nodes for a virtual machine. For this reason, we execute each configuration twice, by swapping the nodes used by the virtual machines, and we compute the average completion time of the two runs.

We can observe that it is nonetheless possible but also efficient to use NUMA policies on several virtual machines, even if the policy to use is different from a virtual machine to the other. For 2 of the 5 tested pair of application, namely `cg.C + lu.C` and `facesim +`

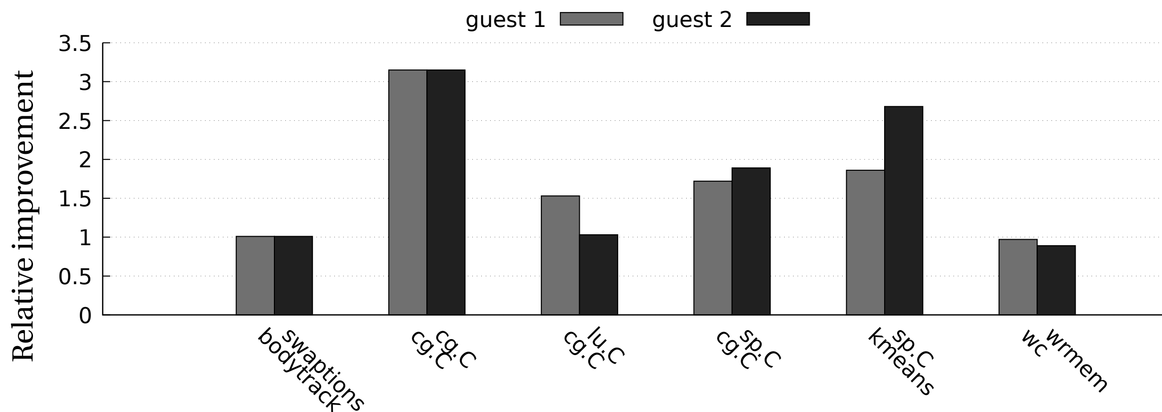

Figure 5.9: Relative improvement of XenNUMA over Xen with 2 consolidated VMs (48 cores each, higher is better)

`lu.C`, one of the applications is neither improved nor degraded while the second one is improved, with up to 300% speedup for `lu.C` when collocated with `cg.C`. For the 3 other pairs, both applications are improved, with a maximum improvement of 440% speedup for `sp.C` when collocated with `cg.C`. Overall, for every application pair we executed, one of the applications is systematically improved by at least 100% and the other application is not degraded.

Consolidated virtual machines

In Figure 5.9, each virtual machine has 48 vCPUs. We pin each vCPU to a single physical CPU in order to avoid performance variations caused by the vCPU placement policy of Xen. With this setting, each physical CPU executes two vCPUs, one belonging to the first virtual machine and one belonging to the second virtual machine.

We can observe that, for consolidated workloads as well, using NUMA policies improve the performance of the applications. As for collocated workloads, two situations occur. In the first case, one of the application is improved while the second one is not degraded, this is the case of `cg.C` consolidated with `lu.C` where `cg.C` is speeded up by 50%. In the second case, both applications are improved. This is the case for the pairs `cg.C + cg.C`, `cg.C + sp.C` and `kmeans + sp.C` with a maximum speedup of 200%. However, a new case shows up where none of the application is improved. This is the case for the pairs `bodytrack + swaptions` and `wc + wrmem`.

5.3 Conclusion

This chapter shows how to implement efficient NUMA memory management policies in a hypervisor. This chapter also evaluates the impact of NUMA management on virtualized application performance. First, we describe the implementation, in the Xen hypervisor,

of the three main NUMA management policies which exist in native configurations: first-touch, round-4K and Carrefour. For two of these policies, first-touch and Carrefour, we encountered limitations to their implementation. For Carrefour, the limitation is inherent to the architecture of Intel processors. For the first-touch policy, the limitation is due to the software design of the Xen hypervisor. Then, we evaluate the NUMA policies we implemented on single and multiple virtual machines configurations. On single virtual machine experiments, we show that, the choice of an appropriate NUMA memory management policy improves the application performance, up to 500%. This performance improvement is the result of the reduction of the virtualization overhead which drops below 50% for 25 of the 29 evaluated applications. This overhead diminution confirms that the virtualization overhead observed with the default NUMA memory management policy of Xen is caused by an ineffective NUMA management. Multiple virtual machines experiments show that, as in native configurations, the usage of an appropriate NUMA memory management policy increases the performance of several virtual machines that share the same hardware.

CHAPTER 6

CONCLUSION

The NUMA architecture is becoming quite common. Even if this kind of hardware remains rare for individual users, it is now pervasive in datacenters. In parallel, individual users as well as industries now use the cloud computing for everyday tasks. The cloud computing relies on virtualization of large and powerfull computers, bundled in datacenters. This simultaneous growing of both the NUMA architecture and system virtualization leads these two technologies, which evolved independently, to now combine. This thesis presents an analysis of the issues this combination involve and possible solutions to tackle them. The first idea it brings is that when executing large virtual machines, which spread on many NUMA nodes, one of the main performance factor is the NUMA management of the virtual machine memory. The second and main idea it brings is that already existing NUMA memory management algorithms perform well even in virtualized configurations and they only need to be implemented in the hypervisor instead of the guest operating system.

We show these ideas by analysing the causes of virtualization performance overhead for a large set of applications from 5 different benchmark suites: Parsec 2.1, Nas Parallel Benchmark 3.3, the Mosbench, X-Stream applications and YCSB. We show that a part of this overhead is due to already studied causes: the I/O virtualization and the IPI virtualization. We then explain how to implement three state of the art native NUMA memory management policies: first-touch, round-4K and Carrefour, in the Xen hypervisor. These policies work differently. We show how to adapt each of these policies to an hypervisor which have no access to the applications. The round-4K policy perform an eager allocation of memory and do not need to worry about applications. The first-touch policy lazily allocates memory and needs to be notified by the guest operating system of some applications requests. The Carrefour policy migrates memory regions after observing accesses patterns using the sampling functionalities of the hardware. We evaluate the performance of virtualized applications using an appropriate NUMA policy and show that the overhead compared to

native execution drops from up to 500% to below 50% for almost all the applications.

Overall, this work answers to the precise question: "Is it possible to efficiently shift the NUMA management from the guest operating system to the hypervisor ?". We clearly show that the answer is yes, but this answer brings new questions. Some of these questions are pratical and short term while some others are more about the large picture. The two following sections cover these two types of question.

6.1 Future works

Our implementation of traditional NUMA policies in a hypervisor is a proof of feasibility. This implementation also allows us to evaluate the benefits of shifting the existing NUMA management to the hypervisor. Our work could however be improved in several ways.

First, the round-1G policy, despite of being the less efficient policy for the applications we evaluate, is the only one to allow an actual usage of large pages to the guest. Designing policies able to provide large pages but still providing an efficient NUMA memory management could increase the performance for TLB miss sensitive applications.

Second, on our implementation, the first-touch policy cannot be used along with the IOMMU. As we stated in Section 5.1, this constraint is caused by the Xen design. An implementation which overcome this constraint could bring better performance for I/O intensive applications which need a first-touch policy.

Third, our implementation of first-touch is efficient enough when the application does not stress the page allocation system of the guest. For more memory allocation intensive tasks, such as the Metis applications: `wc`, `wr` and `wrmem`, our hypercall batching induces an overhead which could be avoided with a better implementation, as we explain in Section 5.2. Using an allocator such as *scalloc* [3] or *llalloc* [4], which only rarely releases pages, should prevent this overhead.

We choose to implement the NUMA memory management inside the hypervisor. We did so because the hypervisor is the only software which should have a complete vision of the machine resources usage and thus is the best suited to manage the memory resource. Another approach is to expose the NUMA topology to the guest and let the state of the art native NUMA policies work in the guest. We believe this approach would be inefficient in a multiple virtual machines scenario as two or more virtual machine could use the same interconnect links and saturate it without noticing, leading the NUMA policies to take inefficient decisions. A comparison between the two approaches could nonetheless be interesting.

Additionally, we could explore the possibility of using NUMA management in the guest, even with a hypervisor that does not cooperate. Indeed, hypervisors that neither expose the NUMA topology to their guest nor perform efficient NUMA management are common. We can ask if the guest applications have no hope to execute efficiently or if there is a way for the guest system to discover the harware details wihout any help of the hypervisor. Chat-

zopoulos et al. suggest with their work on MCTOP [18] that it is possible for the guest to discover the NUMA topology only with performance measurement.

6.2 Perspectives

Our work could be improved in many ways, either performance improvement, adaptation to new hardware features or better cooperation with the guest. This work also brings new questions about virtualized NUMA management. Is our approach the best one ? Can we add more constraints ? Additionally, we can take a step backward and look at the larger picture. In our approach, we shift the NUMA decision process from the guest operating system to the hypervisor. As we explain in Chapter 4, we do so because the hypervisor is the only software layer that has a complete view of the hardware resources utilization. Moreover, we show in Chapter 3 that a guest operating system that uses emulated hardware or take complex decisions, like scheduling, may suffer from performance drops. As a result, it seems not feasable nor desirable to implement resources management in a guest operating system. We can then ask what is the role of an operating system in a virtualized configuration.

Some recent works [47, 15, 16] bring some answers to this question. The unikernel is a specialized kernel compiled along with an application. A unikernel extensively relies on the hypervisor for resource management and isolation. The only protection a unikernel provides come from the compiler that ensures the application makes no malicious operations. Alternatively, Manco et al. propose a less radical approach with Tinyx [48], a build system that creates minimalistic guest systems targeting single applications. In both cases, the operating system and the end user application tend to merge in a single application like software layer. While the resource management that tends to move to the hypervisor and the guest that becomes applications, we can ask is the hypervisor is becoming just another operating system.

BIBLIOGRAPHY

[1] J. L. Abell, J. Fern, M. E. Acacio, et al. Glocks: Efficient support for highly-contended locks in many-core cmps. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 893–905. IEEE, 2011.

[2] A. Agarwal and M. Cherian. *Adaptive backoff synchronization techniques*, volume 17. ACM, 1989.

[3] M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'15*, pages 451–469, 2015.

[4] llalloc: Lockless memory allocator. http://locklessinc.com/.

[5] Amd64 architecture programmer's manual volume 2: System programming. `https://http://support.amd.com/TechDocs/24593.pdf`, 2017.

[6] Amd i/o virtualization technology (iommu) specification. `http://support.amd.com/TechDocs/48882_IOMMU.pdf`, 2015.

[7] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong. I/o paravirtualization at the device file boundary. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 319–332, New York, NY, USA, 2014. ACM.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'03*, pages 164–177, 2003.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'09*, pages 29–44, 2009.

[10] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, pages 29–40. ACM, 2006.

[11] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[12] J.-P. Billaud and A. Gulati. hclock: Hierarchical qos for packet scheduling in a hypervisor. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 309–322. ACM, 2013.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI'10*, pages 1–16, 2010.

[15] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 250–257. IEEE, 2015.

[16] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide. A performance evaluation of unikernels. 2015.

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

[18] G. Chatzopoulos, R. Guerraoui, T. Harris, and V. Trigonakis. Abstracting multi-core topologies with mctop. In *Proceedings of the Twelfth European Conference on Computer Systems-EuroSys" 17*, number EPFL-CONF-227458, pages 544–559. ACM Press, 2017.

[19] L. Cheng, J. Rao, and F. C. M. Lau. vscale: Automatic and efficient processor scaling for smp virtual machines. In *Proceedings of the European Conference on Computer Systems, EuroSys'16*, pages 2:1–2:14, 2016.

[20] Y. Cheng, W. Chen, X. Chen, B. Xu, and S. Zhang. A user-level numa-aware scheduler for optimizing virtual machine performance. In *International Workshop on Advanced Parallel Processing Technologies*, pages 32–46. Springer, 2013.

[21] Y. Cheng, W. Chen, Z. Wang, and X. Yu. Performance-monitoring-based traffic-aware virtual machine deployment on numa systems. *IEEE Systems Journal*, 11(2):973–982, 2017.

[22] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[23] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30:16–29, 2010.

[24] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993.

[25] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'13*, pages 381–394, 2013.

[26] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'13*, pages 33–48, 2013.

[27] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining numa locks. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 65–74. ACM, 2011.

[28] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'12*, pages 247–256, 2012.

[29] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'14*, pages 73–84, 2014.

[30] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu. Extending xen with intel virtualization technology. *Intel Technology Journal*, 10(3), 2006.

[31] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009.

[32] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti. Remix: Online detection and repair of cache contention for the jvm. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 251–265, New York, NY, USA, 2016. ACM.

[33] R. Ganesan, Y. Murarka, S. Sarkar, and K. Frey. Empirical study of performance benefits of hardware assisted virtualization. In *Proceedings of the 6th ACM India Computing Convention*, Compute '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[34] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: a garbage collector for big data on big NUMA machines. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'15*, pages 661–673, 2015.

[35] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[36] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.

[37] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 17:1–17:9, 2013.

[38] Pci-sig sr-iov primer. `https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`, 2011.

[39] V. Kazempour, A. Kamali, and A. Fedorova. Aash: An asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 85–96, New York, NY, USA, 2010. ACM.

[40] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[41] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 64–75. IEEE, 2004.

[42] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. Vmm-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 33–43, New York, NY, USA, 2007. ACM.

[43] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *ACM SIGPLAN Notices*, volume 46, pages 169–180. ACM, 2011.

[44] M. Liu and T. Li. Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads. In *Proceedings of the International Symposium on Computer Architecture, ISCA'14*, pages 325–336, 2014.

[45] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[46] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the European Conference on Computer Systems, EuroSys'16*, pages 1:1–1:16, 2016.

[47] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472. ACM, 2013.

[48] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM.

[49] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'91*, pages 269–278, 1991.

[50] G. E. Moore. Cramming more components onto integrated circuits. In M. D. Hill, N. P. Jouppi, and G. S. Sohi, editors, *Readings in Computer Architecture*, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[51] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[52] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[53] A. Nordal, A. Kvalnes, and D. Johansen. Paravirtualizing tcp. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing Date*, VTDC '12, pages 3–10, New York, NY, USA, 2012. ACM.

[54] H. N. Palit, X. Li, S. Lu, L. C. Larsen, and J. A. Setia. Evaluating hardware-assisted virtualization for deploying hpc-as-a-service. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '13, pages 11–20, New York, NY, USA, 2013. ACM.

[55] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 241–252. IEEE, 2003.

[56] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *Proceedings of the symposium on High Performance Computer Architecture, HPCA'13*, pages 306–317, 2013.

[57] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM, 2006.

[58] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the International Symposium on Memory Management, ISMM'06*, pages 84–94, 2006.

[59] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *ACM SIGPLAN Notices*, volume 36, pages 44–52. ACM, 2001.

[60] L. H. Seawright and R. A. MacKinnon. Vm/370—a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[61] R. Shea and J. Liu. Network interface virtualization: challenges and solutions. *IEEE Network*, 26(5), 2012.

[62] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[63] X. Song, H. Chen, and B. Zang. Characterizing the performance and scalability of many-core applications on virtualized platforms. Technical report, Parallel Processing Institute, Fudan University, 2010.

[64] Pthread mutex lock. `https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=nptl/pthread_mutex_lock.c;hb=HEAD`, 2017.

[65] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 286–297. ACM, 2017.

[66] B. Teabe, A. Tchana, and D. Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the European Conference on Computer Systems, EuroSys'16*, pages 3:1–3:14, 2016.

[67] P. Tembey, A. Gavrilovska, and K. Schwan. Merlin: Application-and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[68] L. van Doorn. Hardware virtualization trends. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 45–45, New York, NY, USA, 2006. ACM.

[69] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.

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

[71] Numa aware scheduling. https://wiki.xen.org/wiki/Xen_4.3_NUMA_Aware_Scheduling, 2015.

[72] Credit scheduler. https://wiki.xenproject.org/wiki/Credit_Scheduler, 2017.