An interface to implement NUMA policies in the Xen hypervisor

Gauthier Voron
Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6

Gaël Thomas
Télécom SudParis, Université Paris-Saclay

Vivien Quéma
Univ. Grenoble-Alpes, Grenoble INP LIG laboratory

Pierre Sens
Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6

Abstract

While virtualization only introduces a small overhead on machines with few cores, this is not the case on larger ones. Most of the overhead on the latter machines is caused by the Non-Uniform Memory Access (NUMA) architecture they are using. In order to reduce this overhead, this paper shows how NUMA placement heuristics can be implemented inside Xen. With an evaluation of 29 applications on a 48-core machine, we show that the NUMA placement heuristics can multiply the performance of 9 applications by more than 2.

1. Introduction

Modern large multicore machines have a complex memory architecture, called Non Uniform Memory Access (NUMA) architecture. Such machines are formed by a set of nodes connected via a high-speed network, called "interconnect". Each node contains several CPUs, a memory bank and a deep cache hierarchy.

Achieving high performance on a NUMA architecture is difficult because the interconnect or a memory controller can easily saturate. This saturation comes from an inadequate memory placement on the NUMA nodes, which leads to many messages on the interconnect or towards a memory controller. Preventing saturation is difficult because, as shown by Lachaize et al. [23] and confirmed by our evaluation, a single memory placement policy on the NUMA nodes is not efficient for all the applications. For this reason, current operating systems provide distinct NUMA placement policies for distinct memory access patterns.

Unfortunately, the Xen hypervisor, which simulates several physical machines through virtual machines, imposes a single, basic NUMA placement policy. As a result, on a set of 29 applications evaluated on a 8-node machine with 48 cores, we measure an overhead caused by an inefficient memory placement of up to 700% with Xen. For 15 of the 29 applications, the overhead is higher than 50%, and for 11 of the applications, the overhead is higher than 100% (see Figure 1).

To decrease this overhead, Amazon EC2, which provides on-the-shelf Haswell multicores with 40 virtual CPUs and 160 GB of memory, proposes to expose the NUMA topology to the virtual machines. The guest operating system is then able to apply a NUMA policy, and hence improve performance. This solution is not satisfactory because it prevents an efficient load balancing of the virtual CPUs on the physical CPUs. Indeed, when the hypervisor migrates a virtual CPU to a physical CPU of a new NUMA node, the hypervisor dynamically modifies the NUMA topology of the virtual machine, which is not supported by any of the current mainstream operating systems.

Instead of exposing the NUMA topology to the virtual machine, we propose to mitigate the NUMA effects at the hypervisor level, by allowing various NUMA policies to be implemented by the hypervisor. Implementing NUMA policies in an hypervisor is challenging. A NUMA policy must decide where to place the memory of a process. It has thus to know which part of the memory is used by a process. But an hypervisor is only aware of the virtual CPUs assigned to a virtual machine, not of the processes or their memory. For this reason, the hypervisor cannot understand which part of the memory belongs to which process, and can thus not easily implement an advanced NUMA policy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…15.00

DOI: http://dx.doi.org/10.1145/3064176.3064196

In this paper, we propose to enable the efficient implementation of NUMA policies inside the Xen hypervisor with a new simple interface. This interface consists in two new hypercalls, i.e., calls from the guest operating system to the hypervisor. The first hypercall is used by the guest operating to inform the hypervisor that it released some memory. The second hypercall is used to select a NUMA policy.

Based on this interface, we show how we can implement four policies. These policies are based on the first-touch, the interleaved and the Carrefour [12] policies (see Section 3). We evaluate the policies using 29 applications from 5 different benchmark suites on a 8-node machine comprising 48 cores using three setups: a setup with a single virtual machine using all 48 cores and two setups with several virtual machines sharing the 48 cores. Overall, our evaluation shows that the NUMA policies we implemented inside Xen can drastically improve the performance of many applications, while hiding the NUMA topology to the virtual machine. More precisely, we found that:

- With a single virtual machine spanning 48 cores, using an efficient NUMA policy divides the completion time of 9 applications by more than 2. The maximum improvement we observe is of 6 times.
- With an evaluation of consolidated workloads of multiple virtual machines, we show that using an adequate NUMA policy also drastically reduces the completion time. Among 11 configurations, an efficient NUMA policy reduces the completion time of at least one virtual machine by more than 2 in 9 cases.
- With a single virtual machine spanning 48 cores, only 4 applications remain degraded by more than 50% when we compare Xen with its best NUMA policy and Linux with its best NUMA policy. This result shows that most of the large overhead observed with Xen on large multicore was actually caused by an inefficient NUMA policy.
- Because of a design choice at the hardware level, it is impossible to use an IOMMU [6] with the first-touch policy in the hypervisor without deeply modifying the I/O stack of the hypervisor.

The remainder of the paper is organized as follows. Section 2 gives some background on Xen. Section 3 describes the NUMA policies studied in the paper. Section 4 presents both the interface and the implementation of the NUMA policies. Section 5 reports the evaluation of the policies. Section 6 discusses related work and Section 7 concludes the paper.

2. The Xen hypervisor

This section provides some background on the Xen hypervisor. An hypervisor simulates several physical machines through virtual machines. A virtual machine holds a set of virtual CPUs (vCPUs), a set of virtual devices, and a memory, called, in Xen, the physical memory of the virtual machine.

2.1 The memory subsystem

An hypervisor has to isolate the memory of the virtual machines. Modern hypervisors ensure isolation by leveraging an *hypervisor page table* provided by the hardware. The hypervisor page table maps the physical memory of a virtual machine to the memory of the machine, called, in Xen, the *machine memory*.[1] An hypervisor activates the hypervisor page table when it switches the processor to *guest mode*, i.e., a special mode that changes the processor behavior to facilitate virtualization. In native mode, an application accesses memory through a single page table that maps a virtual address to a physical address, while in guest mode, the processor additionally translates the physical address of the guest to the machine address using the hypervisor page table.

2.2 The I/O subsystem

In this section, we provide some background on the I/O subsystem, because we later explain why the implementation of one of the NUMA policies (first-touch) has an incompatibility with the virtualized I/O mechanism provided by modern processors.

In Xen, a special virtual machine, called *dom0*, is used to manage the other virtual machines, called *domU* virtual machines. Dom0 can also be used to manage the I/Os for the domUs. In this case, dom0 directly accesses physical devices, while domU virtual machines access physical devices via the dom0 virtual machine.

2.2.1 Para-virtualization technique

Xen implements two complementary techniques to intercept an access to a virtual device of a domU virtual machine. The first technique is the full-virtualization technique: the guest operating system executes a legacy driver, and Xen intercepts the low level I/O operations performed by the driver. The second technique is the para-virtualization technique: the guest operating system executes a modified driver, which calls the hypervisor to perform the I/O. In our experiments, we use the para-virtualized drivers implemented in Linux because it yields better performance.

2.2.2 The PCI passthrough driver

When a domU virtual machine accesses a virtual device, Xen intercepts the access, forwards the access to dom0, which executes the request, and gives the result to the domU virtual machine. When a guest operating system starts a Direct Memory Access (DMA), it uses a physical address that has to be translated into a machine address before the access. On

[1] In the remainder of the text, we use the Xen terminology to describe the different memory levels. In other hypervisors, e.g., in KVM, an address of the machine memory is called a system physical address (SPA) or a host physical address (HPA), and an address in the physical address space of a virtual machine is called a guest physical address (GPA).

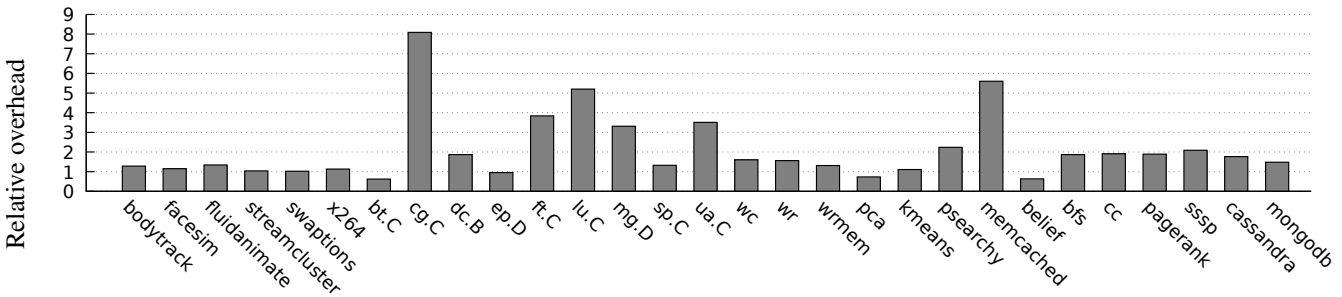

Figure 1. Relative overhead of the Xen compared to the Linux (lower is better).

the 48-core machine we used for the experiments, we have measured that reading a 4 KiB block (with the `O_DIRECT` flag to avoid caching) takes 307 μs, while it only takes 74 μs in Linux.

Instead of letting the hypervisor perform the address translation, modern processors provide an IOMMU component [6]. This component is a memory management unit that can be directly used by devices to translate a physical address into a machine address. With an IOMMU component, a device accesses the physical memory without involving the hypervisor, thus significantly reducing the large overhead mentioned above.

Xen is able to use an IOMMU with the PCI passthrough driver. However, the PCI passthrough driver is relatively restrictive. While the AMD IOMMU can associate devices to virtual machines at the device granularity, the PCI passthrough driver associates devices to virtual machines at the PCI express bus granularity. Fortunately, the machine used in our experiments has two PCI express buses. For this reason, we can reserve a PCI express bus for a domU virtual machine. Dom0 can then use the other PCI express bus. With this setting, we have measured that reading a 4 KiB block takes 186 μs (still 74 μs in Linux). 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.

3. NUMA policies under study

This section presents the NUMA architecture with the NUMA policies evaluated in the paper and a performance evaluation showing that all the presented policies are useful.

A NUMA architecture is composed by a set of NUMA nodes. Each NUMA node may contain a memory bank, I/O controllers and CPUs. The hardware statically partitions the machine address space into NUMA regions. Each CPU uses a map associating each region to a single NUMA node in order to transparently route memory accesses to the appropriate NUMA nodes.

A NUMA placement policy, or simply NUMA policy, is in charge of choosing on which NUMA node to place each virtual address used by a process. In native Linux systems, NUMA policies rely on the page table. More precisely, a NUMA policy maps a virtual address of a process to a NUMA node by mapping the virtual address to a physical page that belongs to the NUMA node.

As highlighted by several research studies [12, 17, 26], a NUMA placement policy should both (i) balance the load on all the memory controllers in order to avoid overloaded memory controllers, and (ii) enforce memory access locality in order to avoid the saturation of the interconnect.

We propose to study four NUMA policies. Three policies (round-1G, first-touch and round-4K) are static policies: they initially map a virtual address to a NUMA node, and they do not dynamically change the initial mapping. The last policy (Carrefour) is a dynamic policy: Carrefour can dynamically remap a virtual address to a new NUMA node in order to improve the average latency of memory accesses. The round-1G policy is the only (and thus default) NUMA policy implemented in Xen. The first-touch, round-4K and Carrefour policies are NUMA policies that have been implemented for the Linux operating system.

3.1 The first-touch policy

The first-touch policy is the default NUMA policy used in Linux. More precisely, Linux uses a lazy memory allocation policy. When it creates a new virtual address space, Linux does not physically allocate the pages. When a thread of the process accesses a page for the first time, the access is thus invalid. Linux intercepts the invalid access and maps the virtual address to a physical page.

With the default first-touch policy, Linux allocates the physical page from the NUMA node of the thread that performs the first access. If the NUMA node does not have enough free pages, Linux allocates the memory from another node selected using a round-robin policy.

The first-touch policy is often efficient because the thread that accesses a page for the first time is often the thread that performs most of the accesses to the page. This is typically the case for the pages corresponding to thread stacks, but

also for the pages that host a data structure only accessed by the thread that allocated the structure.

The first-touch policy is, however, particularly inefficient if a single thread allocates and initializes the memory for all the other threads. This is typically the case of applications implementing a master-slave pattern, in which a master thread prepares the memory for the slaves. In such applications, because of the first-touch policy, the master thread maps a large part of the memory from its NUMA node when it prepares the memory. When the slaves execute, they perform most of their memory accesses to this NUMA node, yielding contention and poor performance.

3.2 The round-4K policy

Linux also proposes a policy that we call round-4K. This policy allocates physical pages of 4 KiB and selects NUMA nodes on which to allocate these pages using a round-robin policy. With this policy, memory accesses are usually well-balanced on all the NUMA nodes, which presents the advantage of balancing the load on memory controllers. However, the round-4K policy has the drawback of inducing a large number of remote memory accesses.

For applications that tend to saturate a single memory controller with the first-touch policy, typically, applications designed with a master-slave pattern, the round-4K induces better performance than the first-touch policy. In such applications, the round-4K policy trades the saturation of a single node with a possible saturation of the interconnect links, which yields lower memory access latencies [12, 17].

3.3 The round-1G policy

Xen uses a default NUMA policy, that we call the round-1G policy. Xen eagerly allocates the physical memory of a virtual machine during its creation. Xen tries to pack the memory and the vCPUs of the virtual machine on the minimal number of underloaded NUMA nodes by reserving a physical CPU per vCPU.[2] These NUMA nodes form the *home-nodes* of the virtual machine.

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. Then, 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 physical GiBs of a virtual machine are always fragmented.

3.4 The Carrefour policy

Carrefour [12] is a dynamic memory placement policy for NUMA architectures executing the Linux operating system. Carrefour dynamically migrates pages in order to increase the memory access locality, while avoiding contention on memory controllers and interconnect links.

Carrefour monitors the memory access patterns of the threads using hardware counters. It implements three heuristics to migrate or replicate the hottest physical pages, i.e., the most accessed physical 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. Carrefour also implements two heuristics that it triggers when it detects that the interconnect saturates. The *migration heuristic* migrates 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. In our study, we have discarded the replication heuristic, because it has only a marginal effect on performance, and because implementing this policy within the Xen hypervisor would require radical changes in the design of the Xen memory manager.

3.5 A case for each NUMA policy

As presented in the previous section, Linux offers several policies. However, their usefulness remains questionable: are all these policies important to achieve good performance? If the answer to this question is no, offering an interface to implement several NUMA policies in Xen is useless. If the answer to this question is yes, it probably makes sense to implement the same NUMA policies within the Xen hypervisor.

3.5.1 Evaluation of the NUMA policies in Linux

In order to verify that all the NUMA policies are useful, Figure 2 reports the improvement in terms of completion time relative to the default first-touch policy when using the different NUMA policies in Linux. The Figure presents the evaluation of 29 applications on AMD48, a 48-core machine with 8 NUMA nodes, running a native Linux operating system (see Section 5 for a detailed description of the hardware and software setting). We exhaustively evaluate all the possible combinations of static and dynamic policies implemented in Linux: first-touch, first-touch/Carrefour, round-4K and round-4K/Carrefour.

We can first observe that the NUMA policy has a huge impact on performance for many applications. 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 can 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`). This result answers the above question: it is probably worth implementing within Xen the NUMA policies that are available within Linux.

[2] `http://wiki.xen.org/wiki/Xen_4.3_NUMA_Aware_Scheduling#Soft_Scheduling_Affinity`.

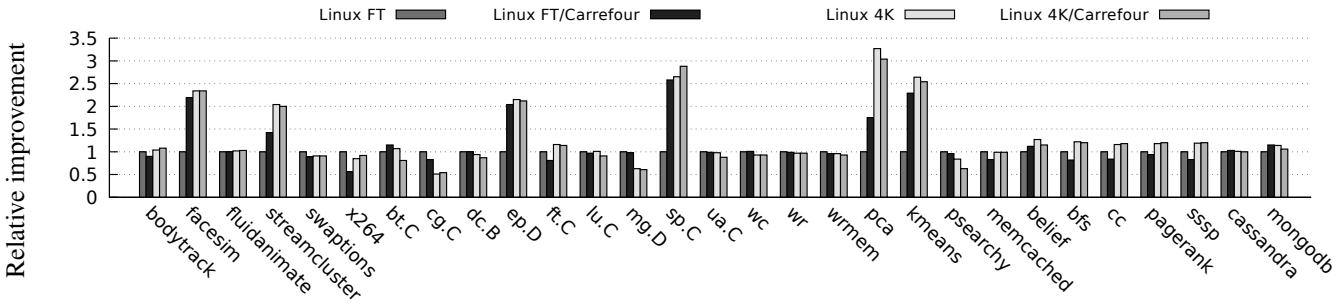

Figure 2. Improvement of the completion time of various NUMA policies in Linux on AMD48 with 48 threads relative to the first-touch policy (higher is better).

	Load imbalance		Interconnect load		Imbalance level
	First-touch	Round-4k	First-touch	Round-4k	with first-touch
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 1. Effect of the static NUMA policies in Linux on AMD48 with 48 threads.

3.5.2 Analysis of the behaviors

In order to understand why all the policies are useful, Table 1 reports two metrics measured with the first-touch and the round-4K policies. The table does not report the metrics when executing Carrefour because Carrefour already uses all the available performance counter registers. The 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.[3]

We can classify the applications in three groups (column imbalance level of Table 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 3.1, for these applications, the first-touch policy is perfect, because each thread tends to mostly access data structures that is 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 3.2, 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 1). Carrefour tends to improve their performance because it improves their memory access locality (the interconnect load is

[3] The hardware counters actually give a metric which vary 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.

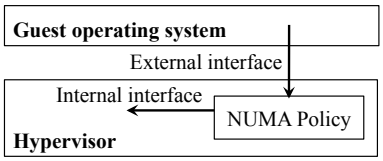

Figure 3. External and internal interfaces.

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 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.

4. Implementing NUMA policies inside Xen

As all the NUMA policies evaluated in the previous section are useful in Linux, we have implemented all these policies in Xen. In order to implement the NUMA policies inside Xen, we have identified three important needs. First, each of the policies needs a mechanism to map a virtual page of a process to a machine page of any NUMA node. Second, the Carrefour policy needs a mechanism to dynamically migrate a virtual page of a process to a new NUMA node. Third, the first-touch policy needs a mechanism to trap the first access of a process to a virtual page.

These needs have driven the definition of the interface we propose in this paper. As presented in Figure 3, the interface can be split into two parts: the external interface is used by a NUMA policy to communicate with the guest operating system, whereas the internal interface is used to communicate with the hypervisor. As we wanted to minimize the modifications performed in the code of the guest operating system, we tried to minimize as much as possible the size of the external interface.

4.1 The internal interface

The internal interface consists in two functions. The first function implements the mechanism to map the virtual page of a process to a NUMA node. The second function implements the mechanism to migrate the virtual page of a process to a new NUMA node.

In an hypervisor, directly placing the virtual address of a process to a NUMA node is difficult because an hypervisor executes virtual machines, not processes. A guest operating system maps a virtual page of a process to a physical page of a virtual machine. Modifying this mapping from within the hypervisor is difficult because an hypervisor cannot easily know which physical page of a virtual machine is used or not. Moreover, the hypervisor cannot easily synchronize with the operating system in order to avoid concurrent accesses to the guest page table.

For these reasons, we have chosen to implement the memory placement mechanism by leveraging the hypervisor page table. In this setting, a NUMA policy lets the guest operating system map a virtual page to any physical page. Then, the NUMA policy maps the physical page to a NUMA node by mapping the physical page to a machine page of the node.

For the same reasons, the migration mechanism relies on the hypervisor page table. It starts by write protecting the entry of the physical page in order to avoid concurrent writes on the copied page. Then, the migration mechanism copies the page to its new location and updates the entry of the physical page in the hypervisor page table.

4.2 The external interface

The external interface consists in two functions. The first function is used to select a NUMA policy. The second function is required for the first-touch policy to trap the first access to a virtual page performed by a process.

4.2.1 Selecting the NUMA policy

An administrator selects a NUMA policy for a whole virtual machine because Xen is not aware of the processes inside the virtual machine. Notice that, for this reason, an administrator must not colocate different processes with contradictory NUMA access patterns inside the same virtual machine.

By default, a virtual machine boots with the round-4K policy. We do not provide an interface to dynamically switch to the round-1G policy because, as presented in the evaluation (see Section 5.3.3), the round-1G policy is much less useful than the other policies. Instead, we provide an option to boot a virtual machine with the round-1G policy that can be used for testing purposes.

Then, when the virtual machine runs, Xen provides a new hypercall to dynamically change the NUMA policy of a virtual machine. This hypercall can switch to the first-touch policy and activate/deactivate the Carrefour policy.

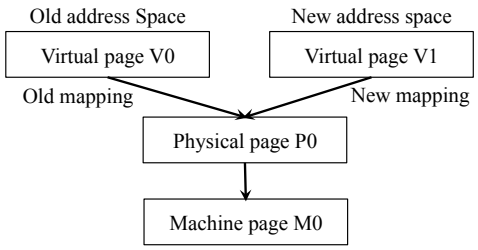

Figure 4. Issue to implement first-touch in Xen.

4.2.2 Trapping the first access to a page

The second hypercall is used to trap the first access to a page. It is used by the guest operating system to communicate a queue of recently allocated and released physical pages. We introduce this function because of a mismatch: the first-touch policy has to trap the first access of a process to a virtual page, but the hypervisor manipulates physical pages that belong to a virtual machine, not to a specific process.

This mismatch is problematic because the hypervisor is not involved when the guest operating system releases a physical page and reallocates the page to a new process. Figure 4 illustrates the problem. An hypervisor can easily trap the first access performed by a virtual machine to a physical page P0 by letting the entry of P0 empty in the hypervisor page table. In this case, during the first access, the hypervisor will map P0 to a machine page, for example M0. However, when the operating system reallocates P0 from an old virtual page V0 to a new virtual page V1, the hypervisor is not involved. The first access to P0 through V1 does not generate an hypervisor page fault: in the best case, it only generates a guest page fault, which is directly caught by the guest operating system, without involving the hypervisor.

4.2.3 An hypercall for each page release

We implement the first-touch policy by invalidating, in the hypervisor page table, the entries of the physical pages that are not used by the guest operating system. When the guest operating system allocates the physical page to a process, the first access of the process generates an hypervisor page fault. We use then this hypervisor page fault to implement the first-touch policy.

In order to implement this mechanism, the hypervisor has to know when a process of a guest operating system releases a physical page. We might want to use the ballooning driver to get that knowledge. The ballooning driver is used by guest operating systems to release pages to the hypervisor, which can then give them to other virtual machines. However, when a guest operating system releases a page through the ballooning driver, the guest can no longer use that page. In our case, the guest operating system has to be able to reallocate the free page to a new process at any time, which precludes using the ballooning driver.

As the ballooning driver is inadequate, we have chosen to use a para-virtualization technique by introducing a new hypercall. The guest operating system triggers this hypercall when it adds a page to its free list, in order to inform Xen that the page is not used anymore. However, calling the hypervisor for each page release is inefficient for an application that releases physical pages often. This is for example the case of several Mosbench applications that we evaluate. These applications rely on the Streamflow allocator [34] because it better scales than the default glibc allocator when the number of cores increases. Unfortunately, the streamflow allocator continuously calls the Linux `mmap`/`munmap` functions to manage the memory. As a result, a Mosbench application such as `wrmem` releases a physical page every 15 μs. Even executing an empty hypercall in Xen for each page release already divides by 3 the performance of `wrmem`. This large slowdown can only annihilate the performance improvement brought by the first-touch policy.

4.2.4 Batching the hypercalls

We can easily solve the problem of the hypercall cost by batching the hypercalls. Instead of calling the hypervisor when it releases a page, the guest operating system accumulates several free pages in a page queue, and sends the whole queue after several page releases.

However, batching the hypercalls 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 is maybe already 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 costly.

We solve the problem by trapping both the page allocation and release of the guest operating system. At high level, we define a global queue shared by all the cores and protected by a lock. 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 a NUMA policy 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 NUMA policy 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 costly 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 [20].

4.3 Implementation of the NUMA policies

With the interfaces provided to NUMA policies, implementing the first-touch, the round-4K and the Carrefour policies is straightforward. We have implemented the round-4K policy in Xen by statically allocating 4 KiB pages in a round-robin fashion from the virtual machine's home nodes when the virtual machine is created. This implementation relies on the internal interface (see Section 4.1). For the first-touch policy, we rely on the hypercall provided by the external interface (see Section 4.2): when the guest operating system releases a page, the NUMA policy simply invalidates the entry in the guest page table.

In order to implement the Carrefour policy, we have ported the code of Carrefour in Xen[4]. The original Carrefour implementation defines two components: the system component and the user component. The system component runs in the Linux kernel. It gathers the low-level hardware counters and associates metrics to the hot pages. It also defines a system interface to migrate a page from one node to another and to read the metrics. The user component uses these metrics to choose which page to migrate and to select the destination node.

In Xen, the user component executes as a process in the dom0 virtual machine. The system component executes inside Xen. Instead of using the operating system interface to communicate with the system component, the user component performs an hypercall, which is trapped by Linux and forwarded to Xen. Then, instead of observing the memory accesses performed by the threads of the processes, the system component observes the memory accesses performed by the vCPUs of the virtual machines. In order to migrate a page, the system component relies on the internal interface (see Section 4.2).

4.4 Limitation

The implementation of our first-touch policy has two limitations.

[4] The code of Carrefour is available at `https://github.com/Carrefour`.

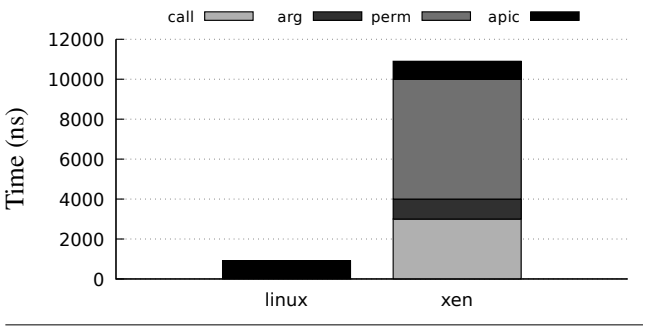

Figure 5. IPI cost repartition

4.4.1 Incompatibility between first-touch and the IOMMU

Our implementation is incompatible with the IOMMU mechanism [6] (see Section 2.2). The IOMMU translates the physical addresses of a virtual machine into the machine addresses in order to perform transparent DMA transfers directly to the guest virtual machine memory. However, during this translation, the IOMMU cannot handle invalid entries in the hypervisor page table.

If an entry is invalid, the IOMMU aborts the transfer and indicates that an I/O error happened. Unfortunately, because of a hardware design choice, the IOMMU asynchronously notifies the hypervisor that an error occurred. As a result, the hypervisor may handle this error after the guest operating system did. In this case, even if the hypervisor maps a machine page in the hypervisor page table, it is too late: the operating system already considered that the I/O was impossible and returned an error to the process that performed the I/O.

In order to trap the first access to a physical page, the first-touch policy invalidates entries in the hypervisor page table. If the physical page is used by the guest operating system as a DMA buffer, the I/O becomes impossible. For this reason, when we evaluate the first-touch policy, we disable the IOMMU.

4.4.2 Content of the released pages

Our first-touch implementation has a second small limitation. Before a page release, Linux fills the page with zeros. For this reason, the free pages in Xen are all equivalent with the same content. The first-touch policy can thus map any page to any physical address of the guest. However, this behavior could be a limitation for an operating system that stores per-page housekeeping data in free pages.

5. Performance evaluation

This section presents an evaluation of the NUMA policies implemented inside Xen. We first present the hardware and then the software setting. We also present Xen+, an improved version of Xen used in the evaluation, along with its evaluation. Then, we compare the different NUMA policies

		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
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. Behavior of the applications.

in Xen+ with both single and multiple virtual machines. Finally, we present a comparison of Linux and Xen+ when we use the best possible NUMA policy for each application in both Linux and Xen+.

5.1 Hardware setting

We perform our evaluation on an AMD machine, called AMD48 hereafter. AMD48 has 8 NUMA nodes with 6 CPUs/16 GiB per node. In total AMD48 has thus 48 cores and 128 GiB of RAM.

The exact memory system of AMD48 is described in [5]. In summary, AMD48 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 machine handles L3 cache misses with the HT3 (HT-assist) protocol. 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 of the dom0 virtual machine 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.

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 3. Cache and memory access latency on AMD48.

Table 3 reports the time to access the caches and the memory on AMD48. 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-hop access in the uncontended case). On the contrary, we can 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).

5.2 Software setting

For each experiment presented in this section, we report the average of 6 runs. We use the Linux 3.9 kernel, along with gcc 4.6.3, libgomp 3.0 and glibc 6. For the analysis, we have selected applications often used to measure the performance of large multicores [7, 12, 13, 33]. We evaluate 29 applications from the Parsec 2.1 (precompiled version), the NPB 3.3 (openMP version), the Mosbench benchmark (Streamflow allocator [34]), a set of X-Stream applications [33], and the YCSB benchmark [11] on Cassandra and MongoDB.

For the Xen experiments, we use Xen 4.5. We hide the NUMA topology to the guest. The dom0 virtual machine is pinned to the CPUs of node 0. As we can only select the NUMA policy for a whole virtual machine, we only run a single benchmark application per virtual machine in all the experiments.

5.3 Xen+

In order to highlight the effect of the NUMA policies in Xen, we use an improved baseline, called Xen+, in which we mitigate other well-know virtualization costs: the cost of virtualized I/Os and some cost of virtualized inter-processor interrupts.

5.3.1 Virtualized I/O cost

One of the virtualization bottleneck is caused by I/Os (disk or network) [1, 18, 24, 29]. We partially remove this overhead by activating the IOMMU with the PCI passthrough driver (see Section 2.2).

However, as presented in Section 4.4, we cannot activate both the IOMMU and the first-touch policy. For this reason,

we only activate the PCI passthrough driver for the round-4K and round-1G policies.

5.3.2 Virtualized IPI cost

Virtualizing inter-processor interrupts (IPIs) is another well-known cause of virtualization overhead. As presented in Figure 5, in native mode, sending an IPI is relatively fast (0.9 μs). However, in guest mode, sending an IPI takes much more time (10.9 μs).

As reported by Ding et al. [16, 35], this overhead is especially detrimental for applications that frequently leave the processor because they wait for an event, e.g., a lock, a condition variable or a network packet. When a thread waits for an event, the thread goes to the sleep state through a context switch. The CPU of the thread may then become idle when no more threads are ready. When the CPU becomes idle, Linux halts the processor up to the next interrupt. In this case, when the event arrives, Linux wakes up a sleeping CPU by sending an inter-processor interrupt (IPI).

As presented in Table 2, we can observe that 7 applications do intentionally frequently leave the CPU (more than 10,000 intentional context switches each second). These applications may suffer from the cost of virtualized IPIs if the CPU goes to the sleep state. Ding et al. propose to solve the problem with a complex algorithm that can handle consolidated workloads.

For the sake of simplicity, we have rather implemented a much simpler solution that only works for non-consolidated workloads. Our solution consists in implementing the pthread mutex and condition variable functions with a spin loop using the MCS algorithm [28]. Thanks to this modification, when a thread synchronizes through a lock or a condition variable, it does not leave the processor. Using a spin loop is, of course, not a long-term solution: we only use this technique to better focus on the NUMA placement problem by eliminating an unrelated, well-known virtualization bottleneck.

This modification significantly improves the performance of `facesim` and `streamcluster` (see the evaluation presented in Section 5.3.3). The modification improves `facesim` by 30% and `streamcluster` by 55%. We also measure that, after the modification, these applications generate zero intentional context switch per second. We activate thus this optimization for Xen in the single virtual machine experiments for `facesim` and `streamcluster`. In order to make a fair evaluation, we also replace the pthread mutex and condition variable functions by a spin loop for `facesim` and `streamcluster` in Linux.

5.3.3 Xen+ evaluation

Figure 6 reports the overhead of Xen, Xen+ and Linux, relatively to LinuxNUMA. We define LinuxNUMA as a Linux that uses MCS locks for `facesim` and `streamcluster`, and that systematically uses the best possible Linux NUMA policy for each application (Table 4 recalls this best Linux NUMA policy based on the evaluation presented in Figure 2 of Section 3.5).

	LinuxNUMA	Xen+NUMA
bodytrack	Round 4K / Carrefour	Round 4K / Carrefour
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	Round-4K	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	First-Touch	Round-1G
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 4. Best NUMA policies.

By construction, Xen+ is better than Xen, and LinuxNUMA is better than Linux. Figure 6 confirms this result.

When we compare LinuxNUMA with Xen+, we can observe that the overhead of Xen+ relatively to LinuxNUMA remains large: 20 of the 29 evaluated applications still exhibit an overhead higher than 25%, 14 applications exhibit an overhead higher than 50%, and 11 applications have an overhead higher than 100%. As we have already removed some overhead caused by I/Os and IPIs in the Xen+, we can suspect that the remaining large overhead is actually often caused by NUMA effects.

When we compare Xen with Xen+, we can observe that `facesim` and `streamcluster` are largely improved by the use of MCS lock. We can also observe that `dc.B`, `bfs`, `cc`, `pagerank`, `sssp` and `mongodb` are largely improved by Xen+. Column `Hard drive` of Table 2, which reports the disk usage, confirms that these applications heavily use the disk, which makes the PCI passthrough driver especially useful.

We can finally observe that, unexpectedly, for some disk-intensive applications (`belief`, `bfs`, `cc`, `pagerank` and `ssp`), Xen+ is slightly better than Linux. We have not found why Xen+ improves performance, but we suppose that Xen+

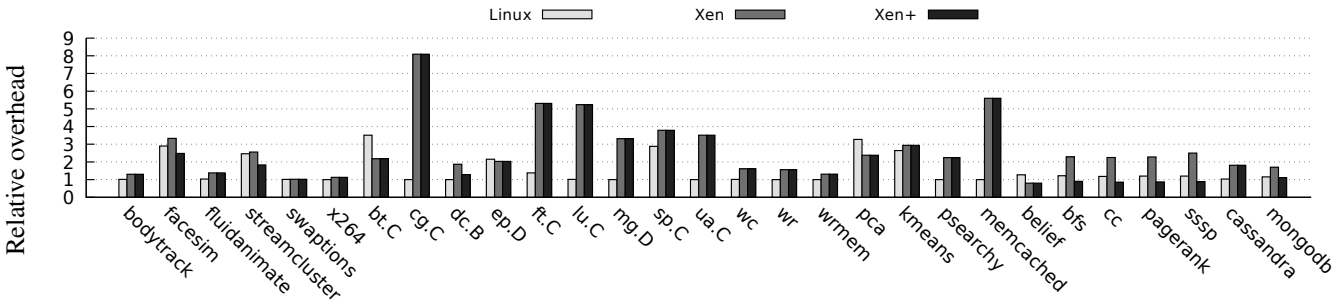

Figure 6. Relative overhead of Linux, Xen and Xen+ as compared to LinuxNUMA (lower is better).

offers a better parallelism during a DMA transfer. In Linux, a DMA buffer is allocated as a contiguous physical address space. This DMA buffer is thus only allocated from a single NUMA node because the hardware partitions the physical address space at coarse grain on the different NUMA nodes. In Xen+, a DMA buffer is distributed on different NUMA nodes thanks to the hypervisor page table, which maps the physical addresses of the guest to memory pages of different NUMA nodes.

5.4 Evaluation of the NUMA policies

In this experiment, we compare the performance of the different NUMA policies of Xen. This evaluation has the goal of showing that selecting an efficient NUMA policy in Xen improves performance. We first evaluate a single virtual machine and then multiple virtual machines.

5.4.1 Single virtual machine

In this experiment, we run a single virtual machine. Each application uses as many threads as the number of vCPUs, and the guest uses as many vCPUs as the number of physical CPUs. We also pin the vCPUs on the physical CPUs and the threads on the vCPUs.

Figure 7 reports, for each of the 29 applications, the improvement relative to Xen+ for each NUMA policy. The column `Xen+NUMA` of Table 4 complements this evaluation by highlighting the best NUMA policies.

We can first observe that using an efficient NUMA policy can drastically improve the performance. 9 applications are improved by more than 100%. In the best case, for `cg.C`, the completion time is divided by 6. Moreover, we can observe that each NUMA policy yields the best performance for some applications. Moreover, the gain brought by each policy is sometimes significant. For example, the first-touch/-Carrefour policy is the best for `bt.C` and it improves performance by 100%, the round-4K/Carrefour policy is the best for `sp.C` and it improves performance by 290%, the first-touch policy is the best for `kmeans` and it improves performance by 170%, and the round-4K policy is the best for `ft.C`, and it improves performance by 315%.

Moreover, we can observe that the default round-1G policy provided by Xen+ is much less efficient than the NUMA policies we have implemented. The round-1G policy is only better than the other policies for four applications (see column `Xen+NUMA` of Table 4). Moreover, as presented in Figure 7, if we replace the round-1G policy by the best other policy, the maximum performance degradation we observe is 10%, and only three applications are degraded by more than 5%.

Finally, we can notice that for the disk-intensive applications improved by the PCI passthrough driver (`dc.B`, `bfs`, `cc`, `pagerank`, `sssp` and `mongodb`, see Section 5.3.3), the first-touch policy seems to systematically drastically degrade their performance as compared to Xen+. This behavior comes from the fact that we disable the PCI passthrough driver when we activate the first-touch policy.

5.4.2 Multiple virtual machines

Figure 8 and Figure 9 present eleven consolidated workloads with two virtual machines. In both experiments, each virtual machine executes a single application with as many thread as available vCPUs in the virtual machine. For each of the applications, we select the best Xen NUMA policy (see column `Xen+NUMA` of Table 4) through the hypercall provided by the external interface. The figures report the improvement of the best NUMA policy over the default NUMA policy on Xen+.

In Figure 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.

In Figure 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

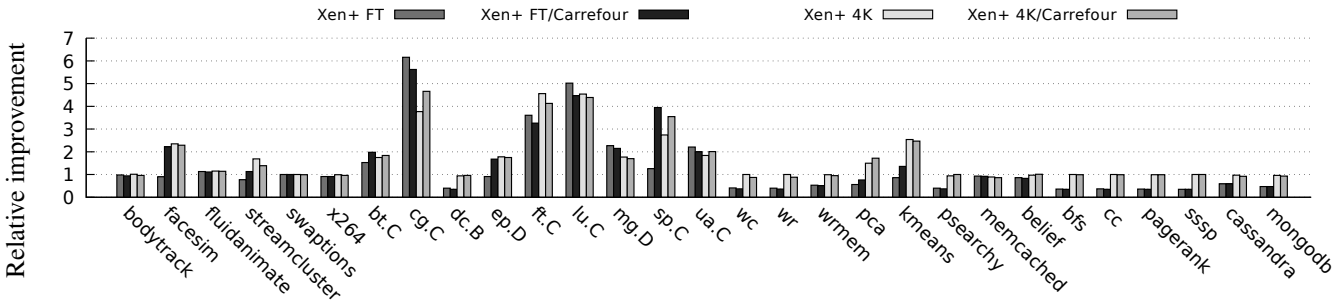

Figure 7. Relative improvement of the NUMA policies in Xen+ as compared to Xen+ (higher is better).

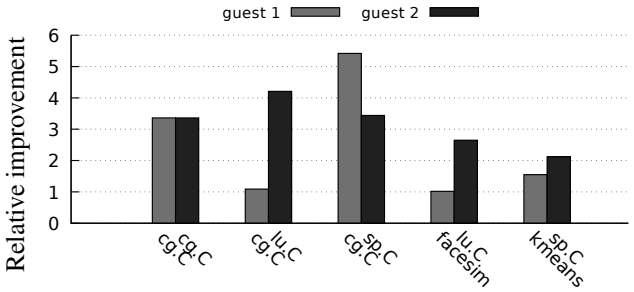

Figure 8. Relative improvement of Xen+NUMA over Xen+ with 2 colocated VMs (24 cores each, higher is better)

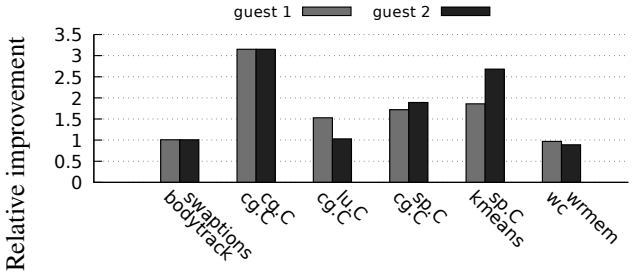

Figure 9. Relative improvement of Xen+NUMA over Xen+ with 2 consolidated VMs (48 cores each, higher is better)

vCPUs, one belonging to the first virtual machine and one belonging to the second virtual machine.

Overall, we can observe that for consolidated workloads, using an efficient NUMA policy also drastically improves performance as compared to Xen+. In the best case (`cg.C` executed with `sp.C` in Figure 8), the performance of the application is improved by 440%. For 9 of the 11 configurations, using an efficient NUMA policy improves performance by more than 50% for at least one virtual machine. We can also observe that only a single configuration (our worst case) is degraded with a better NUMA policy, and by at most 10%. These results confirm that using an efficient NUMA policy is also important for consolidated workloads on large multicores.

5.5 Xen+NUMA versus LinuxNUMA

In this experiment, we compare Xen+NUMA with LinuxNUMA. Xen+NUMA is defined as Xen+ with its best NUMA policy for each application (see column `Xen+NUMA` of Table 4). This experiment has the goal of showing that the large overhead of Xen+ is caused by an inefficient NUMA placement and not by other virtualization overheads. In Xen+NUMA, we use the single virtual machine setting, in which both the threads and the vCPUs are pinned. In LinuxNUMA, we pin the threads to the physical CPUs.

Figure 10 reports the overhead of Xen+NUMA as compared to LinuxNUMA. We can 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 can reduce this overhead, while hiding the NUMA topology to the guest virtual machine.

For the 4 remaining applications, we can observe that `memcached`, `cassandra`, and `ua.C` frequently leave the CPU (column context switch of Table 2). These applications suffer from the cost of virtualized IPIs. They are not corrected by our simple 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) and, despite the use of the IOMMU, may activate a bottleneck in the I/O stack that we have not yet identified.

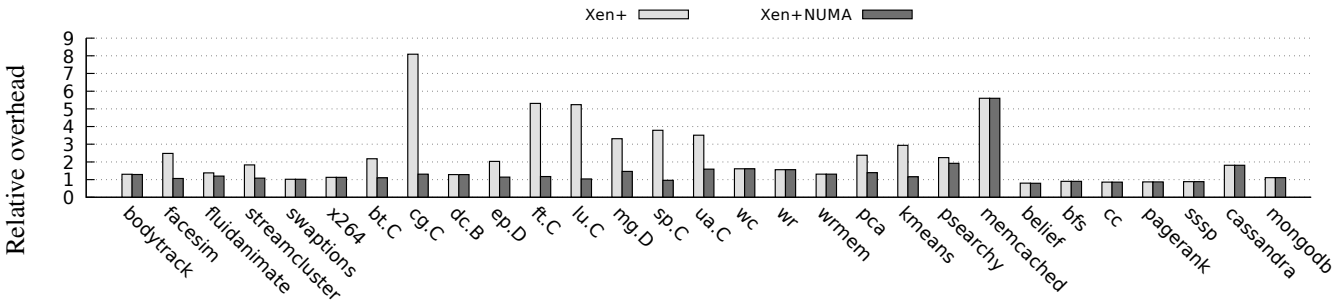

Figure 10. Relative overhead of Xen+ and Xen+NUMA as compared to LinuxNUMA (lower is better).

6. Related work

Many recent works study the performance effect of NUMA architectures: in an hypervisor [21, 26, 32], in an operating system [12, 25], in the locking sub-system [14, 15] or in garbage collectors [17, 38, 45]. These works show that a NUMA architecture can have a dramatic performance impact when either the memory controllers or the interconnect saturates. Our study confirms this observation.

Among the works on NUMA architectures, four ones specifically focus on hypervisors [8, 21, 26, 32]. Overall, none of these works identify the interface to implement several NUMA policies inside an hypervisor. Rao et al. [32] propose the bias random vCPU migration scheduler to place the vCPUs on a NUMA architecture. They show how this algorithm can improve performance (by up to 37%). They show that caches and NUMA locality improves performance when it does not prevent load balancing. Our study is complementary: while Rao et al. are interested by the vCPU scheduling algorithm, we focus on the memory placement policy. Liu et al. propose a static NUMA placement policy when the virtual machine is created [26]. While they provide a single NUMA policy, we show that the hypervisor should provide several NUMA policies. For this reason, in our paper, we identify an interface to implement classical NUMA policies in an hypervisor. Bugnion et al. [8] focus on retrocompatibility. They propose to run legacy (non-NUMA in 1997) operating systems on NUMA architectures by hiding the NUMA architecture using an hypervisor. As for Liu et al., they provide a single NUMA placement policy, while we focus on the interface required to implement several NUMA policies inside the hypervisor. Han et al. [21] present a study of the performance impact of NUMA architectures in an hypervisor, but they do not propose NUMA placement policies.

Several research works are interested in understanding [10, 31] and mitigating interference between several virtual machines: to isolate I/Os performance [9, 19, 30, 36, 42, 43], to enforce the performance of interrupt handling [2], to enforce the performance of blocking locks, [2, 9, 36], to colocate compatible workloads in the same server [22, 27, 44], to avoid cache interference from other virtual machines [36] or to prevent the preemption of the lock holder in case of spinlocks [37, 39–41]. Our work is complementary. We study another performance aspect of virtualization: the effect of NUMA architectures.

Regarding IPIs, we discuss the Ding et al. algorithm [16] in Section 5.3.2 and, for the sake of simplicity, implemented a simpler mechanism to mitigate the IPI cost in non-consolidated workloads. Our evaluation confirms that virtualized IPI is a large source of overhead for some applications on large multicores.

For I/Os, their cost in a virtualized environment has been studied in different research works [1, 18, 24, 29]. Solutions exist to mitigate this cost, especially the work of Gordon et al. [18], which is able to entirely remove the hypervisor from the I/O path. By using the IOMMU, we remove some of the I/O bottleneck in order to better highlight the NUMA effects.

7. Conclusion

This paper shows how we can implement the classical NUMA policies of operating systems inside an hypervisor, while hiding the NUMA topology to the virtual machines. We identify a small interface with two functions between the guest operating system and the hypervisor that enables these implementations. Our evaluation shows that we can significantly improve the performance of 9 of the 29 evaluated applications by more than 100%. We show that we can reduce the overhead of virtualization to less than 50% for most of the evaluated applications.

Our work opens several perspectives. First, except the default round-1G policy, the NUMA policies presented in this paper only consider small pages of 4 KiB. Handling large pages in order to decrease the number of TLB misses should further improve performance. Second, our implementation may exhibit an overhead when an application often releases pages (see Section 4.2.4). Using an allocator such as *scalloc* [3] or *llalloc* [4], which only rarely releases pages, should prevent this overhead. Third, we have identified an incompatibility between the IOMMU and the first-touch policy. Finding an algorithm that prevents the invalidation, in the hypervisor, of pages used by the guest for DMA transfers

is a promising perspective. Finally, automatically selecting the most efficient NUMA policy in an hypervisor or in an operating system remains an open subject.

Availability

The code and the results are freely available at `https://github.com/gauthier-voron/xen-tokyo/tree/carrefour-improved` for the Xen host and `https://github.com/gauthier-voron/linux-xen-ft` for the modified Linux guest.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'06*, pages 2–13, 2006.

[2] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proceedings of the International Symposium on Microarchitecture, MICRO'14*, pages 394–405, 2014.

[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] Cache hierarchy and memory subsystem of the amd opteron processor. `http://portal.nersc.gov/project/training/files/XE6-feb-2011/Architecture/Opteron-Memory-Cache.pdf`, 2011.

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

[7] 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.

[8] 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.

[9] 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.

[10] L. Cherkasova and R. Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'05*, pages 24–24, 2005.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the Symposium on Cloud computing, SoCC'10*, pages 143–154, 2010.

[12] 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.

[13] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the Free-Lunch profiler. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, 2014.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Eli: Bare-metal performance for i/o virtualization. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'12*, pages 411–422, 2012.

[19] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the International Conference on Middleware, Middleware'06*, pages 342–362, 2006.

[20] 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.

[21] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-R. Choi, and J. Huh. The effect of multi-core on hpc applications in virtualized systems. In *Proceedings of the European conference on Parallel processing, EuroPar'10*, pages 615–623, 2010.

[22] Y. Koh, R. C. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'07*, pages 200–209, 2007.

[23] R. Lachaize, B. Lepers, and V. Quema. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'12*, pages 53–64, 2012.

[24] 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 *Proceedings of the international conference on Virtual Execution Environments, VEE'11*, pages 169–180, 2011.

[25] Autonuma: the other approach to numa scheduling. http://lwn.net/Articles/488709/, 2012.

[26] 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.

[27] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 32(3):88–99, 2012.

[28] 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.

[29] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the international conference on Virtual Execution Environments, VEE'05*, pages 13–23, 2005.

[30] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the international conference on Virtual Execution Environments, VEE' 08*, pages 1–10, 2008.

[31] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *Proceedings of the International Conference on Cloud Computing, CLOUD'10*, pages 51–58, 2010.

[32] 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.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'13*, pages 472–488, 2013.

[34] 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.

[35] 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.

[36] 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.

[37] B. Teabe, A. Tchana, and D. Hagimont. The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (i-spinlocks). In *Proceedings of the European Conference on Computer Systems, EuroSys'17*, 2017.

[38] M. M. Tikir and J. K. Hollingsworth. NUMA-aware Java heaps for server applications. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'05*, pages 108–117, 2005.

[39] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the conference on Virtual Machine Research And Technology Symposium'04*, pages 1–14, 2004.

[40] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the International Conference on Parallel Architectures and Compilation, PACT'06*, pages 124–133, 2006.

[41] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the symposium on High-Performance Parallel and Distributed Computing, HPDC'11*, pages 239–250, 2011.

[42] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'13*, pages 243–254, 2013.

[43] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vslicer: Latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *Proceedings of the symposium on High-Performance Parallel and Distributed Computing, HPDC'12*, pages 3–14, 2012.

[44] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the International Symposium on Computer Architecture, ISCA'13*, pages 607–618, 2013.

[45] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In *Proceedings of the International Symposium on Memory Management, ISMM'12*, pages 3–14, 2012.