When eXtended Para-Virtualization (XPV) meets NUMA

Vo Quoc Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, Noel Depalma

When eXtended Para - Virtualization (XPV) Meets NUMA

Bao Bui
IRIT, University of Toulouse, CNRS
bao.bui@enseeiht.fr

Djob Mvondo
Grenoble Alpes University
djob.nvondo@univ-grenoble-alpes.fr

Boris Teabe
IRIT, University of Toulouse, CNRS
boris.teabe@enseeiht.fr

Kevin Jiokeng
IRIT, University of Toulouse, CNRS
kevin.jiokeng@enseeiht.fr

Lavoisier Wapet
IRIT, University of Toulouse, CNRS
lavoisier.wapet@enseeiht.fr

Alain Tchana
I3S, University of Nice, CNRS
alain.tchana@unice.fr

Gael Thomas
Samovar, Telecom SudParis
gael.thomas@telecom-sudparis.eu

Daniel Hagimont
IRIT, University of Toulouse, CNRS
daniel.hagimont@enseeiht.fr

Gilles Muller
lip6, Sorbonne University
gilles.muller@lip6.fr

Noel DePalma
Grenoble Alpes University
noel.depalma@univ-grenoble-alpes.fr

Abstract

This paper addresses the problem of efficiently virtualizing NUMA architectures. The major challenge comes from the fact that the hypervisor regularly reconfigures the placement of a virtual machine (VM) over the NUMA topology. However, neither guest operating systems (OSes) nor system runtime libraries (e.g., HotSpot) are designed to consider NUMA topology changes at runtime, leading end user applications to unpredictable performance. This paper presents e**X**tended **P**ara-**V**irtualization (XPV), a new principle to efficiently virtualize a NUMA architecture. XPV consists in revisiting the interface between the hypervisor and the guest OS, and between the guest OS and system runtime libraries (SRL) so that they can dynamically take into account NUMA topology changes. The paper presents a methodology for systematically adapting legacy hypervisors, OSes, and SRLs. We have applied our approach with less than 2k line of codes in two legacy hypervisors (Xen and KVM), two legacy guest OSes (Linux and FreeBSD), and three legacy SRLs (HotSpot, TCMalloc, and jemalloc). The evaluation results showed that XPV outperforms all existing solutions by up to 304%.

Keywords NUMA, Virtualization

ACM Reference Format:
Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gael Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. 2019. When eXtended Para - Virtualization (XPV) Meets NUMA. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303960

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303960

1 Introduction

Virtualization is the cornerstone of today's computing infrastructures. It relies on a hypervisor, which abstracts away a physical machine into a virtual machine (VM). The concept of hypervisor has been introduced in 1970s [42], revisited twenty years ago with para-virtualization techniques to achieve better performance [7, 9], and simplified ten years ago with dedicated hardware support [28, 31]. However, hypervisors were designed when the mainstream machines were mostly single core. Since that time, machines have evolved to NUMA multicore architectures that offer high performance thanks to a multi-level cache hierarchy and a complex network to connect NUMA nodes that each contains a memory bank and several cores. While the performance promises of NUMA architectures is nearly achieved in bare-metal

systems with heuristics that place the memory and the threads of the processes on the nodes [21, 22, 25, 34], this is not yet the case in virtualized systems.

On a NUMA architecture, current hypervisors are inefficient because they blindly change the NUMA topology of the guest virtual machine in order to balance the load. The hypervisor migrates the virtual CPUs (vCPUs) of a virtual machine when it balances the load on the physical CPUs (pCPUs) or when it starts/stops new virtual machines. The hypervisor also migrates the memory of a virtual machine when it uses ballooning or memory flipping techniques [50]. These migrations change the NUMA topology transparently to the VM [6, 46, 52]. However, guest operating systems (OSes) and their system runtime libraries (SRLs, such as Java virtual machine) are optimized for a given static NUMA topology, not for a dynamic one [51]. Therefore, when the hypervisor changes the NUMA topology of the VM, the guest OS and its SRLs consider a stale NUMA topology, which results in wrong placements, and thus performance degradation.

Existing approaches for virtualizing the NUMA topology fall into two categories, *Static* and *Blackbox*. The *static virtual NUMA (vNUMA)* approach is offered by the major hypervisors (Xen, KVM, VMWare and Hyper-V), and consists in directly exposing to the VM the initial mapping of its vCPUs and VM memory to NUMA nodes when the VM boots. Because the OS and the SRLs are unable to support topology reconfiguration [17, 51], this solution can only be used if the hypervisor fully dedicates a physical CPU (pCPU) to a given vCPU (respectively a machine memory frame to the same guest memory frame), and if this mapping never changes during the lifetime of the VM. This solution is not satisfactory because it wastes energy and hardware resources by preventing workload consolidation. The *Blackbox* approach was proposed by Disco [9], KVM [4], Xen, and Voron et al. [52]. It consists in hiding the NUMA topology by exposing a uniform memory architecture to the VM, and in implementing NUMA policies directly inside the hypervisor. By this way, the *Blackbox* approach can be used in case of consolidated workloads. But, as we experimentally show in Section 2, this approach is inefficient, especially with SRLs. A SRL often embeds its own NUMA policies, which has been proven to be much more efficient than exclusively relying on the OS level NUMA policies [24, 25, 30]. The Blackbox approach nullifies this effort because it hides the NUMA topology from the SRL. Another issue with the Blackbox approach comes from the current implementations of hypervisors, which can make conflicting placement decisions. For instance, we have observed that the NUMA policy of the hypervisor may migrate a vCPU to an overloaded node in order to enforce locality, while the load balancer of the hypervisor migrates back that vCPU in order to balance the load (see Section 6).

In this paper, we propose a new principle called e**X**tended **P**ara-**V**irtualization (XPV) to efficiently virtualize a NUMA architecture. XPV consists in revisiting the interface between a hypervisor and a guest OS and between the guest OS and the SRLs in order to dynamically adapt NUMA policies used in the guest OS and the SRLs when the NUMA topology of the VM changes. XPV extends the well known Para-Virtualization (PV) principle in two directions. First, in the same way as PV actually abstracts away I/O devices [12], XPV extends the principle by also abstracting away the physical NUMA topology into a virtual NUMA topology that can change at runtime. Second, while currently PV is only used at the guest OS level to implement optimized drivers for virtualized environments, XPV extends this principle to the SRLs: we propose to adapt the SLRs with para-virtualization techniques in order to dynamically adapt their NUMA policy when the virtual NUMA topology changes. By doing so, XPV allows each layer in the virtualization stack to implement what it does best: optimization of resource utilization for the hypervisor and NUMA resource placement for the guest OS and the SRLs. Notice that the extension of the PV principle to the application layer is quite new.

The paper also presents a methodology for systematically adapting legacy hypervisors, OSes, and SRLs. We demonstrate that implementing the XPV principle requires modest modifications in these systems. We have implemented the XPV principle in two legacy hypervisors (Xen with 117 LOC changed, and KVM with 218 LOC), two legacy guest OSes (Linux with 670 LOC, and FreeBSD with 708 LOC) and three legacy SRLs (HotSpot Java virtual machine [1] with 53 LOC, jemalloc [2] with 86 LOC, and TCMalloc [30] with 65 LOC). We have evaluated XPV using several reference benchmarks (SpecJBB 2005 [13], BigBench [23], Spec MPI 2007 [14], Spec OMP 2012 [15], and CloudSuite [19]). Using a hardware virtualized with Xen, we compared XPV with four state-of-the-art solutions including Interleaved (the default Xen solution), First-touch (FT) [52], Automatic NUMA Balancing [26], and static vNUMA [48, 49]. The evaluation results showed that XPV outperforms all of these solutions. Firstly, when no topology change is triggered by the hypervisor, XPV outperforms Interleaved, FT and ANB by up to about 130%, 99% and 88% respectively. For some applications, the benefits of XPV is magnified by the optimized NUMA policies embedded into the SRL. For instance, the application milc performs 64% better when TCMalloc (its SRL) is XPV aware. Secondly, when VMs are subject to topology changes, XPV outperforms Interleaved, FT, ANB, and

static vNUMA by up to about 173%, 304%, 88%, and 127% respectively.

In summary, we show that:

- exposing to a VM its actual NUMA topology is the best way to handle NUMA in virtualized systems.
- an adaptable virtual NUMA topology exposition is possible using the XPV principle.
- we can implement XPV in legacy hypervisors, OSes and SRLs with few LOC.
- XPV systematically preserves the performance of the static vNUMA approach when the virtual NUMA topology changes.

Thanks to our contributions, we can thus now run several VMs on a NUMA machine and still use efficient NUMA policies.

The remainder of the paper is organized as follow. Section 2 presents the motivation of our work. Section 3 presents XPV and its implementation methodology. Section 4 presents the implementation of XPV in legacy systems. Section 5 presents the evaluation results. Section 6 presents the related work. Section 7 concludes the paper and presents some future work.

2 Virtual NUMA (vNUMA)

Several research work realized in native environments [21, 22, 25, 34] have demonstrated the necessity to specialize system software (e.g., OS and SRL) with respect to the NUMA architecture. The same observations were made [36, 51, 52] in virtualized environments. Among the existing solutions (presented in Section 6), vNUMA is the most promising one. This section presents vNUMA and the motivations for our work. Although the problem we address in this paper affects all hypervisors (as far from our knowledge), we use Xen for the assessment.

2.1 Description

vNUMA consists in showing to the VM a virtual NUMA topology which corresponds to the mapping[1] of its virtual resources on physical NUMA nodes. By this way, there is no need for the hypervisor to include a NUMA optimization algorithm. At the time of writing of the paper, the most popular hypervisors (Xen, VMware, hyper-V and VMware) implement vNUMA. This is considered as the ideal approach to handle NUMA in virtualized environments because it makes guest OS's NUMA policies (assumed to be the most optimal ones) effective, as argued by VMware in [51][2].

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

[2] "*... Since the guest is not aware of the underlying NUMA, the placement of a process and its memory allocation is not NUMA aware... vSphere 5.x solves this problem by exposing virtual NUMA topology for wide virtual machines.*" by VMware in [51]

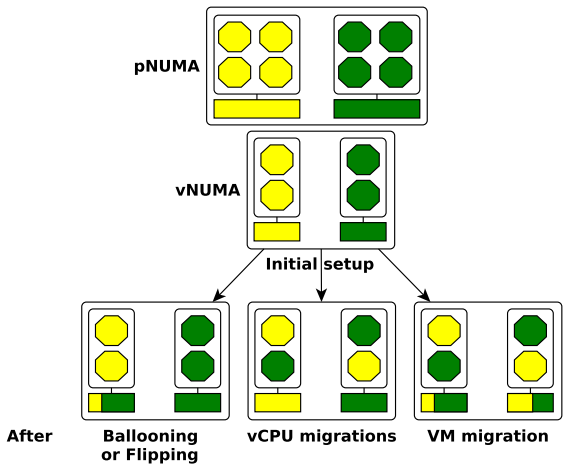

Figure 1. Hypervisor's resource utilization optimizations may lead to vNUMA changes.

2.2 Limitations

The common implementation approach of vNUMA consists of the hypervisor storing the virtual topology of the VM in its ACPI tables, so that the guest OS uses it at boot time as any OS does. This implementation has the advantage to be straightforward. However, its main limitation is that a change in the NUMA topology cannot be taken into account without rebooting the VM [51]. In fact, existing OSes are not designed to dynamically take into account NUMA topology changes. Also, a simple solution based on hot-(un)plug of CPU and memory ressources is not suitable as its might seem, more details in Section 2.4. Topology changes occur in a virtualized system because of resource utilization fairness and optimizations (to avoid waste) implemented by the hypervisor. To achieve these goals, the hypervisor is allowed to dynamically adapt the mapping between the vCPUs and the pCPUs, and the mapping between the guest physical addresses (GPAs) and the NUMA nodes, by changing the mapping between the GPAs and the host physical addresses (HPAs). More precisely, the hypervisor can change the NUMA topology of a VM due to the following decisions (summarized in Fig. 1):

CPU load balancing. Resource overcommitment is the widely used approach for optimizing resource utilization in virtualized datacenters. It consists in allowing more resource reservation than the available resources. For the CPU, this approach could lead to load imbalance, thus unfairness. This issue is addressed by the hypervisor by migrating vCPUs from heavily contended nodes to less contended ones. Such migrations could be frequent due to temporary CPU load imbalance as indicated by VMware [51]. For instance, we have observed that the

execution of three 15vCPUs/12GB SpecJBB 2005 VMs on a 8-node machine - 6 cores per node (two nodes dedicated to the privileged VM in Xen) - generates about 20 vCPUs migrations between different NUMA nodes per minute.

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

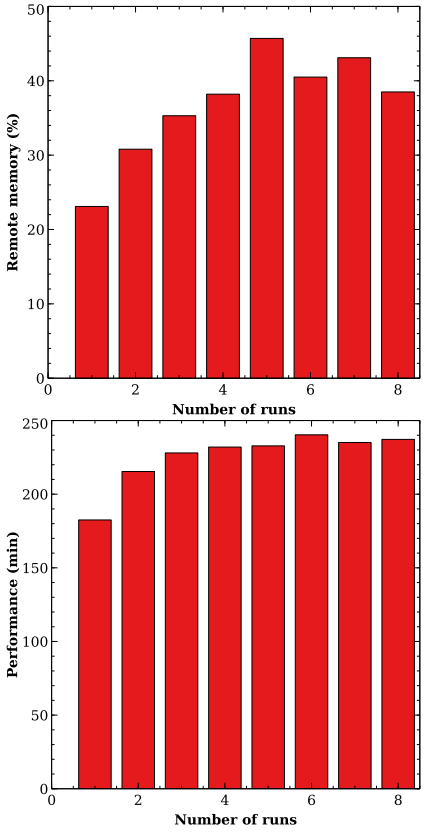

Figure 2. The effect of memory flipping, which occurs on I/O intensive applications. The top curve shows the proportion of the VM's memory which has been remapped on remote nodes after each run. The bottom curve shows the performance of the application after each run.

Memory flipping. Memory flipping is the recent approach used by most hypervisors to implement zero-copy during I/O (e.g., network) operations [3, 11]. It consists in exchanging (instead of copying) memory pages between the driver VM (the one which includes device drivers) and the user VM which sends/receives packets. When the driver VM and user VMs are located on different NUMA nodes (which is commonly the case), user VMs which perform I/O operations will see a portion of their memory remapped on the driver VM's node. To assess this issue, we ran BigBench [23] on our 8-node machine. All the VM resources are initially mapped on a single NUMA node which is distinct from the one used by the driver VM. Without rebooting the VM, we repeated the execution of BigBench 8 times. Fig. 2 shows the amount of remote memory (due to memory flipping) at the beginning of each execution of the benchmark. We can see that it increases in respect with the number of execution, up to 45% of the VM's memory is remote at execution number 5 due to the memory flipping mechanism. This degrades the application performance down to 32%.

VM live migration. VM live migration involves moving a running VM from one physical host to another one. It is a central technology in today's datacenters. For instance, dynamic VM packing (which is a common approach used to optimize resource utilization), physical server failure handling, datacenter maintenance, overheating power supplies and hardware upgrades rely on it [44]. All of this makes VM live migrations very frequent in the datacenter, especially large ones (such as Google Cloud Engine [44]). The migration of a VM may change the mapping of its vCPUs and GPAs on the destination machine. We assess in Xen the impact of using a stale topology as follows. We used two memory intensive benchmarks: STREAM [37] (a synthetic benchmark measuring the sustainable memory bandwidth) and LU from Spec MPI 2007 [14]. The benchmarks run within a VM (called the tested VM) configured with 12 vCPUs and 30GB memory. They use TCMalloc [30] as the SRL. We compared three situations: (1) the VM sees a UMA topology but the hypervisor implements interleave (corresponds to the default Xen solution); (2) the VM sees a NUMA topology which corresponds to its exact resource mapping (noted vNUMA); (3) and the VM sees a NUMA topology which is different from its actual mapping: all of its CPUs are migrated away from the initial nodes (noted Stale vNUMA). The initial resource mapping of the VM is as shown in Fig. 1 while the stale topology is the one caused by the transition labeled "vCPU migration". The experiment results are shown in Fig. 3. Lower is better for LU (the left curve) while it is the opposite for STREAM (the right curve). When we compare Interleaved and vNUMA (first two bars), we can observe that static vNUMA is the most efficient configuration. For instance, we can notice for LU up to 13% of performance difference. However, Interleaved becomes better than vNUMA when the initial topology becomes stale.

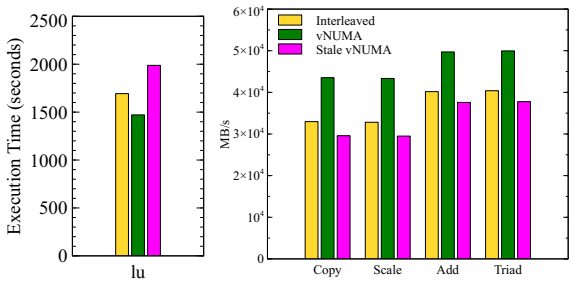

Figure 3. Assessment of the negative impact of presenting a stale topology to the VM when vNUMA is used. Evaluation realized using LU from Open MPI 2007 (left, lower is better) and STREAM (right, higher is better).

2.3 Synthesis

Although vNUMA is the ideal approach for taking into account NUMA in virtualized environments, its current implementation makes it static, thus inefficient facing topology changes. Hypervisor vendors like VMware and Hyper-V have also underlined this limitation. Typically, the VMware documentation [51] states: *"The idea of exposing virtual NUMA topology is simple and can improve performance significantly. [...] On a virtual environment, it becomes likely that the underlying NUMA topology of a virtual machine changes while it is running. [...] Unless the application is properly reconfigured for the new NUMA topology, the application will fail. To avoid such failure, ESXi maintains the original virtual NUMA topology even if the virtual machine runs on a system with different NUMA topology."*. Today, to prevent the inefficiency of vNUMA, hypervisor providers make the following recommendations [16, 20, 29, 32, 40, 45, 47]: (1) either the data center operator enables vNUMA and disables all hypervisor's resource management optimizations, which leads to resource waste or unfairness, (2) either she disables vNUMA and keeps hypervisor's resource management optimizations by using Blackbox/hypervisor-level solutions (as shown above and will be detailed in Section 6), which are not optimal for performance (see Section 5). None of these two solutions is satisfactory.

2.4 Hot-(un)plug as a solution?

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

1. Resource hot-(un)pluging is only possible for CPUs or memory which has been discovered and recorded by the guest OS at boot time [17]. Therefore, the implementation of a hot-(un)plug based solution needs to first show to the VM at boot time the entire physical machine topology, knowing that the actual VM's resource mapping concerns only a subset of the topology. This requires deep kernel code rewriting in both the VM's OS setup code and also the hypervisor code which is responsible for starting a VM. The development cost[3] of this step is very high compared to the approach we present in the next section. Moreover, it would be difficult to port this code to other systems.
2. Assuming that the above is implemented, memory hot-(un)plug is only possible at the granularity of a block (e.g., 512MB in the current Linux kernel version). Remember that topology changes could be caused by the relocation of very few memory pages (e.g., see memory flipping for example).
3. Finally, Hot-(un)plug is not sufficient because the SRL which runs inside the VM is not aware of the new topology.

For all these reasons, hot-(un)plug is considered imcompatible with vNUMA, which is also the VMware opinion [17, 51]. In this paper, we present eXtended Para-Virtualization, a principle for implementing an adaptable vNUMA.

3 eXtended Para-Virtualization

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

3.1 Principle

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

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

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

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

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

3.2 Methodology for making legacy systems XPV aware

In the hypervisor, we implement XPV by adding a notification to the guest kernel when the NUMA topology changes. In the guest kernel, we implement XPV mainly by adapting the NUMA-aware data structures and by forwarding the notification to the SRL when the guest kernel receives a notification from the hypervisor. In the SRL, we implement XPV mainly by adapting the NUMA-aware data structures when the SRL receives a notification from the guest kernel. Fig. 4 presents the components modified to implement XPV, and the remainder of the section details these modifications.

Hypervisor layer adaptation. Today, a hypervisor implements the static vNUMA approach by exposing the NUMA topology through ACPI tables. Using these tables is inadequate for a dynamic NUMA topology because the code that manages these tables in the guest kernel is often intrinsically static. Instead, we implement XPV by exposing a UMA topology through the ACPI tables and by using a new driver, called the `topology manager`. This driver follows the split-driver principle: a part of the driver is implemented in the hypervisor while the other part is implemented in the guest kernel. The hypervisor maintains the NUMA topology of each VM and, when the NUMA topology changes, notifies the VM with an interrupt. Upon notification, the VM retrieves, through a shared memory, its actual NUMA topology (e.g., physical location of the vCPUs).

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

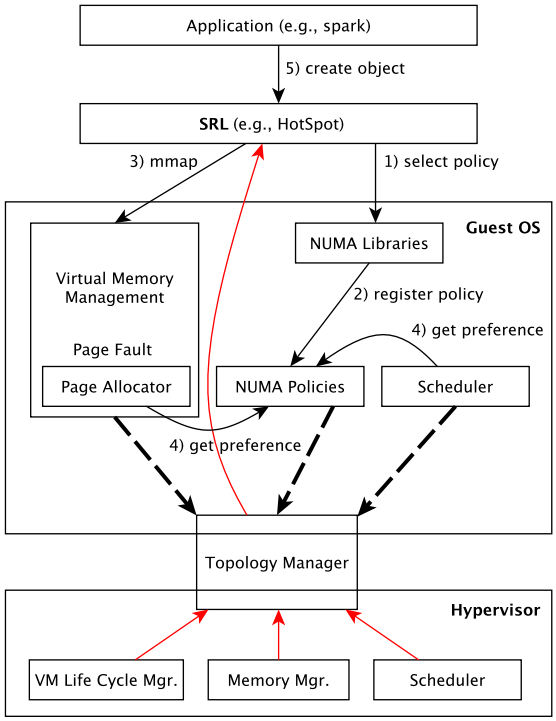

Figure 4. Overview of the components involved in the XPV implementation. Solid black arrows present the different steps (1-5) required to handle a dynamic virtual NUMA topology in SRLs. Dashed arrows show guest kernel components involved when the NUMA topology of the VM changes. Solid red arrows represent the notifications when the NUMA topology changes.

Exposing a UMA topology does not require any modification in the hypervisor because exposing a UMA topology is the default behavior of any hypervisor. Implementing the topology manager is straightforward. It requires modifications in the VM life cycle manager of the hypervisor in order to create and destroy the data structure associated to a VM, in the memory manager in order to record the physical addresses used by a VM and in the scheduler in order to record the location of the vCPUs. These components are easy to identify in any hypervisor and we were able to implement XPV in Xen and Linux KVM.

Guest kernel adaptation. Implementing XPV requires first the implementation of the kernel part of topology manager driver. The kernel uses this driver at bootstrap to retrieve the initial topology of the VM and to initialize its NUMA-aware data structures. Then, when the driver receives a notification from the hypervisor, it increments a counter used by the SRLs to know the current version of the NUMA topology (see below) and updates the

NUMA-aware data structures of three components of the guest OS: the page allocator, the NUMA policy manager and the scheduler. As a consequence of this update, the kernel considers the new NUMA topology for the newly allocated pages and the newly created threads. However, the kernel does not try to relocate the previously allocated pages and threads: instead, we let kernel's NUMA policies (such as Automatic NUMA Balancing in Linux) and the SRL migrate its memory and its thread because we consider that only they can know how to efficiently handle the new NUMA topology.

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

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

Each SRL has to implement its own alogirthm to handle a NUMA topology change. However, we have identified that, in our three studied SRLs (HotSpot [1], TCMalloc [30], and jemalloc [2]), we can apply a systematic methodology to implement XPV. This methodology consists in modifying the code of three components that are often found in SRLs: the bootstrap code, the thread manager and the memory manager. In the bootstrap code, the SRL has to record the initial topology version number of the topology. In the thread manager, the SRL may migrate the threads when the topology changes, i.e., when the topology version number changes. In the memory manager, the SRL has to update its internal data structures when the topology changes, and, if required, to adequately migrate the pages that contain the memory structures already allocated to the application.

4 Technical integration

We applied the methodology described in the previous section in two legacy hypervisors (Xen and KVM), two legacy guest OSes (Linux and FreeBSD) and three legacy SRLs (HotSpot, TCMalloc, and jemalloc).

Table 1 summarizes the efforts required for each legacy system. All the patches are available at [10]. We only present modifications for Xen and Linux, in respect with page length limit.

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

Table 1. XPV integration in several legacy systems.

4.1 Xen modifications

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

In order to implement XPV in Xen, the topology manager first records the initial topology of the VM when the latter boots. This initial topology is stored in a memory region shared between Xen and the guest kernel. Recall that topology changes may happen on the following cases: vCPU migrations (due to vCPU loadbalancing) inside a machine, grants acquisition on new memory pages (due to memory flipping), memory page migrations inside a machine (due to ballooning), and VM migrations between machines. For the three former cases, the modification performed in Xen is straightforward. Whenever a vCPU of a VM is migrated to a different NUMA node, Xen updates the corresponding VM's topology in the shared memory region and notifies the VM by injecting an interrupt in the guest. Regarding page migrations, the new location(s) are taken into account as follows: when the migrated pages return to the guest OS, the latter then examines on which node each page is located and puts the pages into the correct free page lists (see Section 4.2, Memory allocator). The same operation is realized when the VM acquires grants on pages located on new nodes. In fact, no modifications at Xen level are needed in these cases. Concerning changes caused by the migration of the VM, they can be considered as a combination of vCPU migrations and memory migrations. Therefore, we combine the above techniques together for handling such changes.

4.2 Linux modifications

Memory allocator. The major challenge of implementing XPV in an OS is to adapt the page allocator. The question we need to answer is: how does the OS respect the NUMA policy used by applications given that it sees a UMA architecture?

Memory pages on each node are divided into zones. Linux relies on the NUMA policy used by the application to search for free pages. The page allocator applies the NUMA policy to select a node and then a zone in that node for page allocations. Each zone has a component called the buddy system [33] which is responsible for page allocations inside the zone. The buddy system breaks memory into blocks of pages and maintains a separate page list for each block size. Most of the time, allocation requests are for single page frames. In order to get better performance, a per-cpu page frame cache is used to quickly serve these requests. Allocations of multiple contiguous pages are directly handled by the buddy system.

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

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

Automatic Numa Balancing (ANB) [26]. ANB is the most advanced NUMA optimization is Linux. Once activated, ANB periodically unmaps memory pages and then traps page faults when the pages are accessed. By doing this, ANB can know the relation between the tasks and the accessed memory and determine if it should move memory closer to the tasks that reference it. Since migrating memory pages is quite expensive, ANB may moves tasks closer to the memory they are accessing instead. The adaptation of ANB to be XPV aware consists mostly in replacing the default topology information provided by the OS by the actual topology information provided by the hypervisor. Some page fault metrics used by ANB may need to be recalculated when the topology get changed. However, in order to avoid the complexity and also the fact that we don't know how long a topology will last, we decide to not recalculate the metrics in our implementation. Instead, ANB will just continue to work with the newer topology.

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

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

The modification we introduce in HotSpot to make it XPV aware is as follows. At the end of a collection, HotSpot examines the topology version counter (introduced in the system library, see Section 3.2) to check if the NUMA topology is stale. In case of stale topology, HotSpot creates the new group spaces and/or removes the invalid ones according to the new topology and updates the CPU-to-node mapping.

5 Evaluations

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

5.1 Experimental setup

The evaluations are realized on a DELL server having 8 nodes (6 AMD Opteron 6344 cores per node), each linked to a 8GB local memory. Otherwise specified, the

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

We experimented two application categories: those which can use an SRL (Java, C/C++) and those without an SRL (Fortran). For Java, we evaluate SpecJBB 2005 [13] (noted JBB2005) single JVM (the performance metric is the number of *business operations per second* (bops)) and BigBench [23] (the performance metric is the execution time). These applications use HotSpot as the SLR. Regarding C/C++ applications, they are provided with milc and lampps from Spec MPI 2007 [14]. We selected these benchmarks because they are representative of the two categories of Spec MPI applications: medium memory usage and large memory usage. These applications use TCMalloc as the SRL. We also experiment WebServing from CloudSuite [19], which uses the two SRLs and performs a lot of I/O. WebServing is a traditional web service application with four tiers: a web server, a database server, a memcached server and a client. The first three tiers are deployed in the evaluated VM while the client is deployed on a distinct server. The performance metric is the number of operations per second (ops/sec). Otherwise indicated, HotSpot and TCMalloc are launched while enabling their NUMA optimizations. Applications from the second category are bt331, fma3d, swim, mgrid331, applu331 from Spec OMP 2012 [15] and pop2 from Spec MPI 2007.

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

- First-touch (noted FT): this solution is the basic Linux's NUMA management solution. It has been implemented within the hypervisor by Voron et al. [52].
- Automatic NUMA Balancing (noted ANB) [26]: It is the most advanced Linux solution for handling NUMA. KVM naturally includes ANB while VMware implements a similar solution. We implemented ANB in Xen for the purpose of this paper. We did this by reproducing the Linux implementation of ANB in Xen. Such a proactive solution could be seen as the ideal XPV competitor.

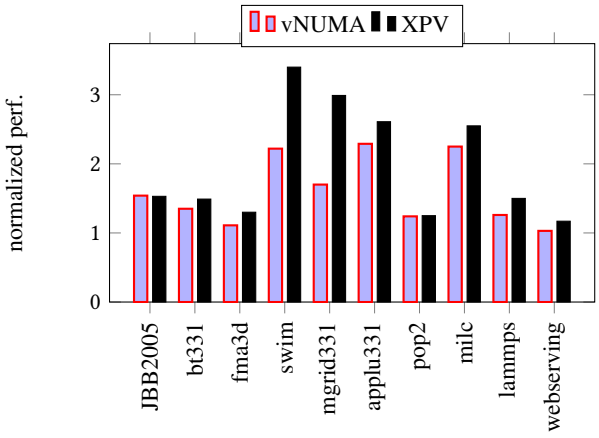

Figure 5. XPV implementation efficiency. XPV is compared with vNUMA (implementer by Xen) when no topology change occurs. The performance gap comes from the fact that the implementation of vNUMA by Xen is less optimized than XPV. Recall that the latter does not use any vNUMA code.

- Interleaved: the VM's resources are packed on the minimum number of NUMA nodes and the memory is interleaved by pages of 1GB (Xen's default policy).
- No policy: the VM's vCPUs and memory are equally distributed over its allocated nodes, no NUMA policy is used.

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

5.2 XPV implementation efficiency

Recall that XPV follows the same approach as vNUMA, which is the presentation of the NUMA topology to VMs. However, the implementation of XPV follows a different strategy, which requires few modifications and make it adaptable (theoretically thus far). One could ask if XPV is at least as efficient as vNUMA when the VM topology stays unchanged. To answer this question, we compared the performance of several applications when they run atop vNUMA and XPV. Recall that the implementation of XPV does not use any Xen's vNUMA code. Fig. 5 presents the evaluation results, interpreted as follows. XPV performs similarly to vNUMA only with JBB2005 and pop2. However, it outperforms vNUMA for the remaining applications, by up to 75% in the case of mgrid331, and even for I/O applications (by up to 15% with webserving) This performance gap between the two solutions, which follow the same static vNUMA approach in this experiment, is explained by the fact that we were able to optimize XPV in comparison with Xen's

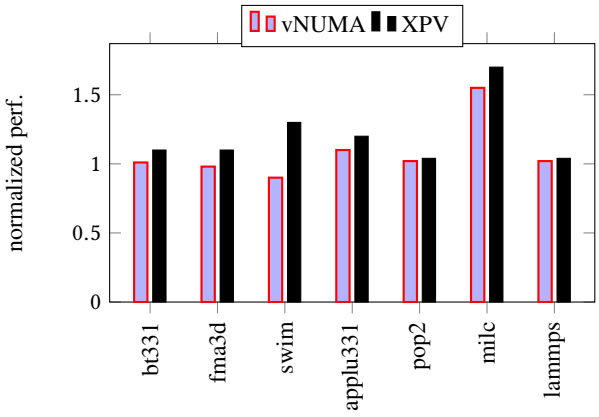

Figure 6. XPV implementation efficiency with small VMs.

vNUMA implementation. Several comments were posted in *Xen-dev* mailing list [18] (a scheduling issue inside the guest), underlying the inefficient implementation of vNUMA.

We also performed the same experiments using smaller VMs (4GB of memory with 4vCPUs). Fig. 6 presents the obtained results. We can observe that XPV outperforms vNUMA with most of the benchmarks (by up to 40% with swim), but the difference is smaller than with bigger VMs (53% with swim for big VMs).

For the remaining evaluations in this paper, instead of using Xen's vNUMA as the static vNUMA baseline, we will use a static version of XPV. Thus, in the rest of this evaluation section, vNUMA refers to a static version of XPV.

5.3 vNUMA vs blackbox solutions

In this section, we compare vNUMA with the existing state-of-the-art blackbox solutions described above. During these experiments, no topology change is triggered by the hypervisor. Every application runs in a VM which uses three NUMA nodes. Fig. 7 presents the evaluation results, interpreted as follows. (1) vNUMA (see "vNUMA with SRL NUMA" bars) outperforms all blackbox solutions. For instance, in the case of swim, vNUMA outperforms Interleaved, FT, and ANB by about 130%, 99%, and 88% respectively.

Concerning JBB2005, which is the only benchmark used by Xen [5] and VMware [51] developers for evaluating the benefits of vNUMA, we can observe a performance gap of about 12% (which is significant) between vNUMA and other solutions. This is compliant with the results obtained by both Xen and VMware developers. However, as mentioned above, the evaluation of other benchmarks shows that vNUMA can bring much more benefits. The main reason which explains the efficiency of the vNUMA approach (the exposition of the topology to the VM) is the fact that it allows optimized NUMA policies implemented by the guest OS and the SRLs (HotSpot and TCMalloc here) to be effective. (2) For some applications, the benefits of vNUMA is magnified by the optimized NUMA policies embedded within the SRL (compare "vNUMA without NUMA SRL" with "vNUMA with NUMA SRL" bars). This is the case for JBB2005, milc and lammps.

For instance, milc performs 64% better when TCMalloc (its SRL) is NUMA aware atop vNUMA. Notice that applications for which "vNUMA without NUMA SRL" equals "vNUMA with NUMA SRL" do not use an SRL. (4) One could imagine that by implementing ANB (which is a Linux NUMA solution) at the hypervisor level, it could provide the same results as using it in a NUMA VM. Our results show that this is not true (compare "ANB" with "vNUMA with NUMA SRL" bars). vNUMA outperforms ANB by up to 88% in the case of swim. This is because in the hypervisor, ANB works at the vCPU granularity, which is not as fine-grained as the thread granularity inside the guest OS. In fact, what a vCPU accesses may suddenly change if the guest schedules another task on it. Therefore, the decision to move a vCPU or a set of memory pages for minimizing remote memory accesses is not precise as it is dictated by several tasks, which is not the case when ANB runs in the guest. (5) Among hypervisor level solutions, ANB is the best one.

5.4 XPV facing topology changes

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

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

- XPV with topology change notifications confined within the guest OS. The SRL level is not informed. This means that only NUMA policies implemented

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

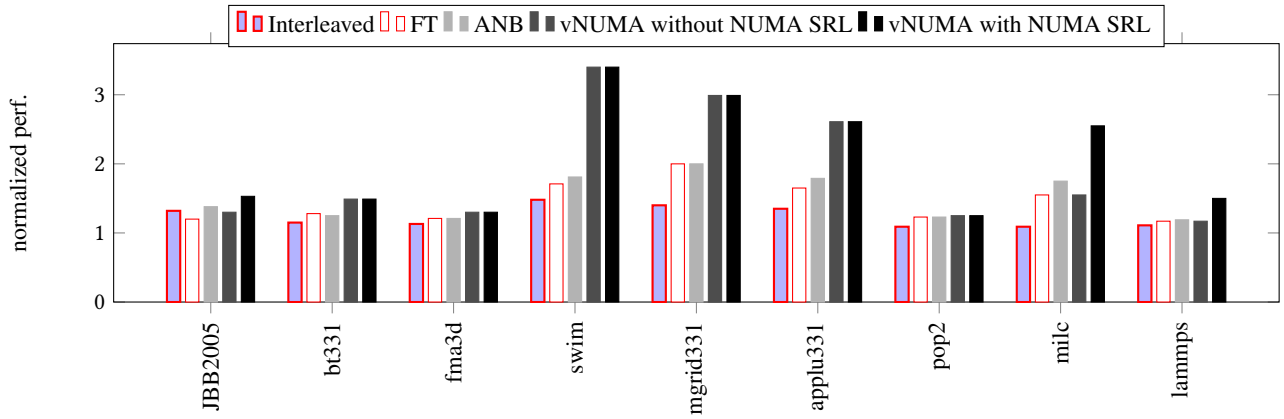

Figure 7. vNUMA compared with the state-of-the-art blackbox solutions (higher is better). This experiment also highlight the importance of NUMA policies embedded within the SRL. No topology change is triggered during this experiment.

within the guest kernel are aware of topology changes. This version is noted "OS only XPV".

- XPV with topology change notifications taken into account by both the guest kernel and the SRL. This version is simply noted "XPV".

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

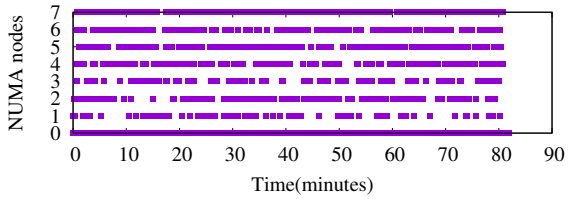

Figure 8. NUMA nodes occupied by $vCPU_0$ of JBB2005 VM during its lifetime when overcommitment is done on the CPU resource.

5.4.1 XPV facing topology changes caused by vCPU loadbalancing

Recall that vCPU loadbalancing could lead to vCPU migrations between different nodes, thus changing the topology of VMs. To create a situation which may lead the hypervisor performing vCPU loadbalancing, we run each tested application in three identical VMs (started

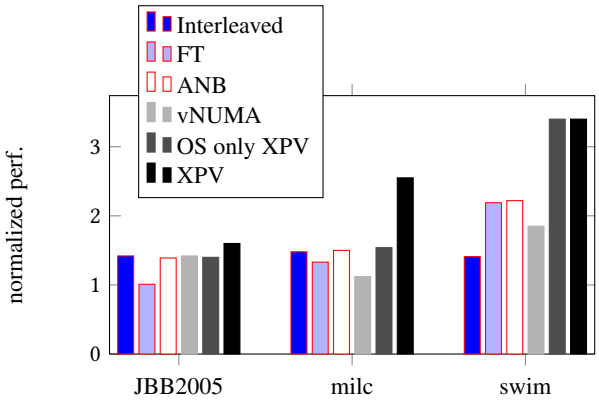

Figure 9. XPV facing vCPU loadbalancing due to the overcommitment of the CPU resource (higher is better).

at the same time). The total number of vCPUs (for the three VMs) is 48 while the number of available cores is 42 (recall that 6 cores are dedicated to the dom0). Thus, the scheduler of the hypervisor is likely to realize vCPU loadbalancing during the experiment. As an example, Fig. 8 shows the different nodes occupied by $vCPU_0$ of JBB2005 VM during its execution. For this application, we counted 1695 topology changes during the execution, corresponding to about 20 topology changes per minutes (noted TC/min). The topology change rates for milc and swim are 5 TC/min and 12 TC/min respectively.

Fig. 9 presents the performance of each application when different NUMA virtualization solutions are used. These results are interpreted as follows. (1) We can see that JBB2005 does not suffer a lot from the issue of topology changes due to vCPU migrations. Except FT, the performance gap between XPV and other solutions (about 12%) is almost the same as when there is no

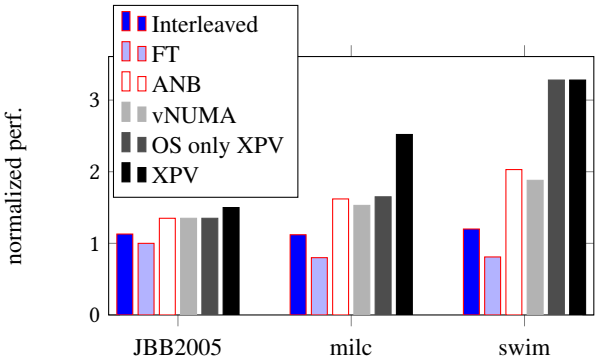

Figure 10. XPV facing memory ballooning (higher is better).

topology change (presented in Fig. 7). FT provides the lowest performance, 58% lower than XPV. In fact, FT only considers NUMA for the first memory allocation operations. Interleaved suffer less because it has interleaved the VM's memory, thus increasing the probability for a vCPU to access a local memory.

Concerning ANB, it enforces memory locality by relocating either vCPUs or memory chunks. (3) Things are different concerning milc and swim. The performance gap between XPV and other solutions is higher in this case (90% and 55% on average for milc and swim respectively). This means that milc and swim are sensitive to topology changes caused by vCPU loadbalancing. (4) The static vNUMA solution significantly degrades the application performance when topology changes are triggered. The performance gap with the adaptable XPV solution is about 12%, 127% and 84% in the case of JBB2005, milc and swim respectively. (5) Making topology changes visible to the SRL layer improves the performance of some applications. The gap between "OS only XPV" and XPV is about 12% for JBB2005 and 65% for milc (recall that swim does not use an SRL). (6) XPV keeps all applications almost to their best performance, which is the one observed when no topology change is triggered (presented in Fig. 7).

5.4.2 XPV facing topology changes caused by memory ballooning

To realize this experiment, we used Badis, a memory overcommitment system presented in [39]. Badis is able to dynamically adjust the memory size of VMs which share the same host in order to give to each VM the exact amount of memory it needs. We ran Badis and our three applications (JBB2005, swim and milc) at the same time. Each VM is launched with 10vCPUs and 20GB which can be adjusted. There is no CPU overcommitment in this experiment.

Fig. 10 presents the evaluation results. The interpretation of these results is almost the same as the one presented in the previous section. The only difference with the previous experiments if the fact that both Interleaved and FT provide very bad performance. For instance, in the case of swim, XPV outperforms Interleaved and FT by about 173% and 304% respectively.

5.5 Automatic NUMA Balancing (ANB) limitations

Due to the inability of existing vNUMA solutions to handle VM topology changes, ANB like solutions have been envisioned as the best compromise thus far. In the performance point of view, our evaluation results confirmed that ANB is the best blackbox solution, although it is largely outperformed by XPV. However, ANB has two side effects which can degrade the performance of the hypervisor. First, ANB decisions can enter in conflict with resource management decisions performed by the hypervisor. For instance, a vCPU loadbalancing decision can move a vCPU to a node, resulting in remote memory accesses, thus the intervention of ANB. The latter will move back the vCPU to its source node, thereby contradicting the previous resource management decision. This issue has also been identified by VMware [51]. Second, a malicious VM can manipulate ANB as follows. Let us consider a VM booted with two vCPUs, one vCPU per node. Let us consider its memory distributed on the two nodes. Even if the VM is presented a UMA topology (as ANB does), an application inside the VM can dynamically discover the distance between vCPUs an a memory chunk using read/write latencies (the STREAM benchmark [37] is the perfect candidate). Therefore, a malicious application can enforce remote memory accesses in order to force ANB in the hypervisor to continually migrate the corresponding VM's vCPUs. These migrations would lead to the migration of tenant VMs, thus impacting their execution. We implemented such a malicious VM and validated this issue.

5.6 XPV internals

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

1. the new interrupt handler in the OS used each time there is a topology update,
2. the new syscall used by the SRL to check the topology version and to retrieve the topology information from the OS (for instance in our experiments, it is called each time the GC runs),
3. the search for a free page on a particular node,

4. the update of memory allocator's data structures (per-CPU caches and the central buddy allocator), and
5. the memory flipping rate.

First, the new interrupt handler in XPV runs in about 368 CPU cycles, which is negligible. Regarding, the new syscalls added to the GC, we found that the GC execution time is not significantly impacted. We rely on virtual dynamic shared object (vDSO) [8], which is a mechanism provided by the kernel for exporting some frequently used read-only syscalls to user-space applications. vDSO routines are called as regular routines, without worrying about performance overhead. Thus, the time consumed by the new syscall is negligible. Second, about the search for a free page, it takes a negligible time to do so as all pages in the kernel are indexed by node (you can refer to Section 4.2) . Third, concerning the update of memory allocator's data structures, whenever a page is freed and returns to the allocator, the kernel examines on which node the page resides and puts it in the correct location. This hardly has any impact on application performance. Finally, we observed a rate of 103 page flips per second for the used benchmarks.

6 Related work

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

6.1 Industrial solutions

Xen [36]. Xen tries to pack the VM's resources on a single node, called the home node. When the VM requires more than one node, Xen proposes both Interleaved and static vNUMA. As shown in this paper, none of these solutions are efficient as XPV.

Oracle VM Server [41]. Oracle VM server proposes the policies used by Xen. It suffers thus the same limitations.

VMWare [51]. VMWare works like Xen, but does not allocate the memory with an interleaved policy. Instead, the VM's memory is simply spread over nodes. Concerning its static vNUMA based solution, VMware is able to update the virtual NUMA topology of the VM by changing the ACPI tables. However, its effectiveness depends on the capability of the guest OS to adapt itself on topology changes. This is not currently the case in the mainstream OSes, which makes the VMware's solution inefficient. In this paper, our XPV solution is able to dynamically adapt the NUMA policies of both the OS and the SRLs when the hypervisor changes the topology.

KVM [4]. KVM implements two blackbox solutions through Linux. These solutions are First-touch (FT) and Automatic NUMA Balancing (ANB). FT is inefficient with SRL based applications while ANB has a lot of limitations as presented in the previous section (conflict with resource management decisions taken by the hypervisor and vulnerable in the point of view of security).

Hyper-V [38]. Hyper-V uses the static vNUMA approach. When it creates a VM, Hyper-V exposes the bootstrap NUMA topology to the VM. Hyper-V does not handle the issues related to topology changes [20].

6.2 Academic solutions

To the best of our knowledge, all academic solutions use a blackbox approach, meaning that the VM sees a UMA topology and the hypervisor implements the NUMA policy. Disco [9] is a hypervisor that enforces locality on a NUMA machine by using page migration and page replication. When the hypervisor observes that a page is intensively used remotely, it migrates or replicates the page. Disco hides the NUMA topology, which makes this solution inefficient for an SRL that implements its own NUMA policy. Similarly, Rao et al. [43], Wu et al. [53], Jaeung al. [27] and Liu et al. [35] proposed new heuristics to place or to migrate the memory or the vCPUs in order to efficiently use the NUMA architecture. However, in their work, they hide the NUMA topology to the VM, which also makes them inefficient for many SRLs. All these solutions are similar to ANB, which has been discussed above.

Instead of proposing new NUMA policies, Voron et al. [52] proposed to implement NUMA policies used in Linux and the Carrefour policy [22] in the hypervisor. The VM can then choose the most efficient NUMA policy. Because a single NUMA policy can not be efficient for all applications, letting an application selects its policy is better than using a single fixed policy. However, the solution proposed by Voron et al. only considers memory (thread placement is not studied). The solution also assumes that all applications inside a VM use the same NUMA policy, which is not the best strategy if a VM runs several processes. Moreover, the solution hides the NUMA topology to the VM, which makes it inefficient for many SRLs. Finally, the solution requires a lot of engineering efforts, while we show that XPV only requires a modest engineering effort.

6.3 Synthesis

As shown in previous sections, all existing solutions fail to virtualize NUMA efficiently. With XPV, we propose to make the static vNUMA approach dynamic. XPV exposes to the guest OS and its SRLs the exact actual NUMA topology. By doing so, XPV allows each layer in

the virtualization stack to do what it does best: resource utilization optimization for the hypervisor and NUMA management for SRLs, helped by the guest OS.

7 Conclusion

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

Acknowledgement

The authors acknowledge the anonymous reviewers for their comments. A special thanks to our shepherd Michael Swift for his great job.

References

[1] [n. d.]. The HotSpot Group. http://openjdk.java.net/groups/hotspot/

[2] [n. d.]. jemalloc memory allocator. http://jemalloc.net/

[3] [n. d.]. Memory flipping in kvm. https://wiki.osdev.org/Virtio

[4] [n. d.]. What is Linux Memory Policy? https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt

[5] [n. d.]. Xen on NUMA Machines. https://wiki.xen.org/wiki/Xen_on_NUMA_Machines visited on September 2018.

[6] Emmanuel Ackaouy. 2006. The Xen Credit CPU Scheduler. http://www-archive.xenproject.org/files/summit_3/sched.pdf

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, 164–177. https://doi.org/10.1145/945445.945462

[8] Daniel Pierre Bovet. 2014. Implementing virtual system calls. https://lwn.net/Articles/615809/ visited on September 2018.

[9] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, 143–156. https://doi.org/10.1145/268998.266672

[10] Bao Bui. 2018. https://github.com/bvqbao/numaVirtualization

[11] Ludmila Cherkasova and Rob Gardner. 2005. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, 24–24. https://www.usenix.net/events/usenix05/tech/general/full_papers/short_papers/cherkasova/cherkasova.pdf

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

[13] Standard Performance Evaluation Corporation. 2005. SPECjbb2005 (Java Server Benchmark). https://www.spec.org/jbb2005/

[14] Standard Performance Evaluation Corporation. 2007. SPEC MPI 2007. https://www.spec.org/mpi2007/

[15] Standard Performance Evaluation Corporation. 2012. SPEC OMP 2007. https://www.spec.org/omp2012/

[16] Frank Denneman. 2016. Decoupling of Cores per Socket from Virtual NUMA Topology in vSphere 6.5. http://frankdenneman.nl/2016/12/12/decoupling-cores-per-socket-virtual-numa-topology-vsphere-6-5/

[17] Frank Denneman. 2017. Impact of CPU Hot Add on NUMA scheduling. http://frankdenneman.nl/2017/04/14/impact-cpu-hot-add-numa-scheduling/ visited on July 2018.

[18] Dario Faggioli. 2015. PV-vNUMA issue: topology is misinterpreted by the guest. https://lists.xenproject.org/archives/html/xen-devel/2015-07/msg03241.html

[19] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, 37–48. https://doi.org/10.1145/2150976.2150982

[20] Aidan Finn. 2015. Hyper-V Dynamic Memory Versus Virtual NUMA. https://www.petri.com/hyper-v-dynamic-memory-versus-virtual-numa

[21] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, 231–242. https://www.usenix.org/system/files/conference/atc14/atc14-paper-gaud.pdf

[22] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM* 58, 12 (Nov. 2015), 59–66. https://doi.org/10.1145/2814328

[23] A. Ghazal, T. Ivanov, P. Kostamaa, A. Crolotte, R. Voong, M. Al-Kateb, W. Ghazal, and R. V. Zicari. 2017. BigBench V2: The New and Improved BigBench. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1225–1236. https://doi.org/10.1109/ICDE.2017.167

[24] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 229–240. https://doi.org/10.1145/2451116.2451142

[25] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[26] Mel Gorman. 2012. Automatic NUMA Balancing. https://lwn.net/Articles/523065/ visited on September 2018.

[27] Jaeung Han, Jeongseob Ahn, Changdae Kim, Youngjin Kwon, Young-Ri Choi, and Jaehyuk Huh. 2011. The Effect of Multi-core on HPC Applications in Virtualized Systems. In *Proceedings of the 2010 Conference on Parallel Processing (Euro-Par 2010)*.

Springer-Verlag, 615–623. http://dl.acm.org/citation.cfm?id=2031978.2032063

[28] Intel. [n. d.]. Intel Virtualization Technology (Intel VT). https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html

[29] Manish Jha. 2015. Whats New in vSphere 6.0 - vNUMA Enhancements. https://alexhunt86.wordpress.com/2015/05/16/whats-new-in-vsphere-6-0-vnuma-enhancements/

[30] Patryk Kaminski. 2012. NUMA aware heap memory manager. https://developer.amd.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf

[31] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf visited on July 2018.

[32] David Klee. 2016. VMware vSphere 6.5 breaks your SQL Server vNUMA settings. https://www.davidklee.net/2016/11/29/vmware-vsphere-6-5-breaks-your-sql-server-vnuma-settings/

[33] Kenneth C. Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624. https://doi.org/10.1145/365628.365655

[34] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, 277–289. https://www.usenix.org/system/files/conference/atc15/atc15-paper-lepers.pdf

[35] M. Liu and T. Li. 2014. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 325–336. https://doi.org/10.1109/ISCA.2014.6853224

[36] Wei Liu and Elena Ufimtseva. 2014. vNUMA in Xen. https://events.static.linuxfound.org/sites/events/files/slides/vNUMA%20in%20Xen_XenDev.pdf

[37] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report.

[38] Microsoft. 2016. Deploying Virtual NUMA for VMM. https://technet.microsoft.com/en-us/library/jj628164.aspx

[39] Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, and Hrachya Astsatryan. 2018. Working Set Size Estimation Techniques in Virtualized Environments: One Size Does Not Fit All. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1, Article 19 (April 2018), 22 pages. https://doi.org/10.1145/3179422

[40] OpenStack. [n. d.]. Hyper-V vNUMA enable. https://specs.openstack.org/openstack/nova-specs/specs/ocata/implemented/hyper-v-vnuma-enable.html

[41] Oracle. 2017. Optimizing Oracle VM Server for x86 Performance. http://www.oracle.com/technetwork/server-storage/vm/ovm-performance-2995164.pdf

[42] Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* 17, 7 (July 1974), 412–421. https://doi.org/10.1145/361011.361073

[43] J. Rao, K. Wang, X. Zhou, and C. Xu. 2013. Optimizing virtual machine scheduling in NUMA multicore systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 306–317. https://doi.org/10.1109/HPCA.2013.6522328

[44] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. 2018. VM Live Migration At Scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. ACM, 45–56. https://doi.org/10.1145/3186411.3186415

[45] Josh Simons. 2011. vNUMA: What it is and why it matters. https://octo.vmware.com/vnuma-what-it-is-and-why-it-matters/

[46] Boris Teabe, Alain Tchana, and Daniel Hagimont. 2016. Application-specific Quantum for Multi-core Platform Scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, Article 3, 14 pages. https://doi.org/10.1145/2901318.2901340

[47] Davoud Teimouri. 2017. NUMA And vNUMA - Back To The Basic. https://www.teimouri.net/numa-vnuma-back-basic/#.WuSuy9axXCJ

[48] Elena Ufimtseva. 2013. [Xen-devel] [PATCH RESEND v2 0/2] xen: vnuma introduction for pv guest. https://lists.xenproject.org/archives/html/xen-devel/2013-11/msg02565.html visited on July 2018.

[49] Elena Ufimtseva. 2013. [Xen-devel] [PATCH v4 0/7] vNUMA introduction. https://lists.xen.org/archives/html/xen-devel/2013-12/msg00625.html visited on July 2018.

[50] VMware. 2009. Understanding Memory Resource Management in VMware ESX Server. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/perf-vsphere-memory_management.pdf

[51] VMware. 2013. The CPU Scheduler in VMware vSphere 5.1. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-cpu-sched-performance-white-paper.pdf visited on April 2018.

[52] Gauthier Voron, Gaël Thomas, Vivien Quéma, and Pierre Sens. 2017. An Interface to Implement NUMA Policies in the Xen Hypervisor. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 453–467. https://doi.org/10.1145/3064176.3064196

[53] S. Wu, H. Sun, L. Zhou, Q. Gan, and H. Jin. 2016. vProbe: Scheduling Virtual Machines on NUMA Systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 70–79. https://doi.org/10.1109/CLUSTER.2016.60