Memory flipping: a threat to NUMA virtual machines in the Cloud

Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, Noel de Palma

Memory flipping: a threat to NUMA virtual machines in the Cloud.

Djob Mvondo
LIG
University Grenoble Alpes
Grenoble, France
barbe-thystere.mvondo-djob@univ-grenoble-alpes.fr

Boris Teabe
IRIT
University of Toulouse
Toulouse, France
boris.teabedjomgwe@enseeiht.fr

Alain Tchana
I3S
University of Nice
Nice, France
alain.tchana@univ-cotedazur.fr

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

Noel De Palma
LIG
University Grenoble Alpes
Grenoble, France
noel.de-palma@univ-grenoble-alpes.fr

***Abstract*—vNUMA is the most recent technology used by hypervisors to deal with Non Uniform Memory Access (NUMA) machines, which currently composed most datacenters. vNUMA consists in presenting to the virtual machine (VM) the initial mapping (at boot time) of its virtual resources to physical resources. By this way, all NUMA optimizations implemented by almost all VM's OS (e.g. Linux) can become effective. However, in order to be effective itself, vNUMA imposes that the initial resource mapping of the VM should remain unchanged during the VM lifetime. Current hypervisors enforce this requirement by avoiding virtual resource migration (between different NUMA nodes, in the same machine), VM migration (between different machines), and memory ballooning.**

However, we found that memory flipping - the most efficient network virtualization approach - violates the above requirement. In other words, a VM which performs network operations leads the hypervisor implicitly performs memory page migrations. In this paper, we show that violating this requirement can degrade performance by up to 18%. We present two solutions which mitigate the issue. We prototype these solutions in Xen hypervisor, a popular open source hypervisor, which is widely used by Amazon Web Services. The evaluation results, performed with well known benchmarks, show that our two solutions are able to almost cancel the issue, while keeping memory flipping effective.

I. Introduction

Virtualization is an essential technology in the Cloud. It allows the sharing of hardware resources between multiple users while enforcing isolation between them. Cloud users benefit from an isolated environment in the form of virtual machines (VMs). This is possible thanks to the hypervisor which is a software layer responsible for the management and sharing of hardware resources between all VMs. In Para-Virtualization (noted PV) and Hardware-Assisted Para-Virtualization (PV-HVM) models[1], a privileged VM (that we call *privVM*, known as Dom0 in Xen [3]) embeds I/O drivers (network and disk) and services as a proxy for user VMs (that we call *unprivVM*, known as DomU in Xen) when they send/receive I/O requests. This is called the *split driver model* in the literature. It works as follows.

To illustrate, let us consider the reception of a network packet. Every *unprivVM* includes a fake device driver (called *frontend*). The latter is linked to a virtual driver (called *backend driver*) located in the *privVM*. Once the packet is received by the network card, the real driver located in the *privVM* is informed. The real device driver identifies the destination VM and informs the backend driver linked to the frontend driver of that destination VM. The backend then synchronizes with the frontend for sending the packet to the VM. As one could see, this path is too complex and has been subject of several optimization. The most important among them, which has been adopted by almost all hypervisor vendors, consists in avoiding packet copy between the *privVM* and *unprivVM*. The current approach to do so is memory flipping.

Memory flipping consists in exchanging page ownership between the *unprivVM* and the *privVM*. This

[1]PV and PV-HVM are the widely used virtualization models because the provide the best performance.

means that the *privVM* gives to the target VM the pages that contain the requests, while the *unprivVM* answers by transferring the right of other memory pages to the *unprivVM*. Memory flipping is currently implemented by almost all hypervisors. For illustration, we use Xen hypervisor, the most popular hypervisor. However, the contributions described in the paper can be easily implemented in other hypervisor. The memory flipping mechanism provides a significant performance benefit for I/O applications but has a significant disadvantage for VMs on NUMA architectures.

Currently, almost all servers in data centers are equipped with a NUMA architecture. A NUMA architecture relies on a complex memory topology with both a multi-level cache hierarchy and a complex network to connect NUMA nodes which each contains a memory bank and several cores. To ensure good performance with such a topology, the OSes must ensure that the applications always run on the node hosting their data to avoid remote memory access. Virtualization does not ignore this and proposes options such as vNUMA for managing NUMA for VMs. vNUMA is an option in Xen and VMware which allows the NUMA topology (including the location of memory pages on NUMA nodes) of a VM to be presented to its guest OS. In other words, NUMA topologies of VMs are virtualized and presented to them at startup. Therefore, guest OSes are aware of the NUMA topology and can therefore avoid remote memory access. For vNUMA to be efficient, the presented topology of a VM must be static and unchanged during the VM lifetime. This is because today OSes do not yet support a dynamic NUMA topology. The hypervisor usually disables for VMs using vNUMA all mechanisms likely to modify their topology: VM migration and memory ballooning.

We noticed that memory flipping can be the cause of topology modification for VMs. Indeed, the widespread configuration on a NUMA topology dedicates an entire node to the *privVM* to isolate it from other *unprivVMs*. Frequent network communications between the *privVM* and a *unprivVM* (the backend driver and the frontend driver) lead to a significant exchange of pages between the two VMs, so a displacement of the memory of the *unprivVM* from its initial nodes to the *privVM* node and vice versa. Note that after the frontend drivers release the pages containing the request data, those pages can be used by other applications in the *unprivVM*, causing a remote memory access for these applications and therefore leading to degraded performance. The solution provided in Xen for this is to use memory copy during the communication between the backend driver and the frontend driver. But, this solution is not suitable because many memory copies lead to disastrous performance for I/O applications in *unprivVMs*.

In this paper, we present two solutions which address this issue due to memory flipping for VM hosted on NUMA architectures. These solutions are able to almost cancel remote memory access induced by memory flipping, thus ensuring optimal applications performance.

- The first solution establishes a pool of pages in the *unprivVM*, and these pages are dedicated to memory flipping and cannot be used by other applications in the *unprivVM*. By doing so, we ensure that only pages in the pool can be remote, and this prevents any remote access from applications in the *unprivVM*.
- The second solution is to keep the current communication mechanism as it is. But periodically, we asynchronously repatriate pages (from our *unprivVM*) that have been transmitted to the *privVM*.

We implemented our solutions within the Xen hypvervisor and evaluated these solutions with well-known benchmarks. Our evaluation shows that our solutions are able to almost cancel remote memory access induced by memory flipping without resorting to memory copy, thus ensuring optimal applications performance. In the rest of the paper, Section II presents an overview of virtualization and NUMA handling virtualization. Section III presents the motivations of the work. Section IV contains our solutions followed by the evaluation in Section V. Section VI presents the related works and finally we conclude in Section VII.

II. Background

A. Virtualization

Virtualization technologies are becoming more and more popular. Nowadays it is hard to imagine data centers, web hosting and other e-infrastructures without virtualization. The main advantage of virtualization technologies is optimal hardware utilization. Several instances of virtual machines can be run on the same physical host, therefore improving resource utilization. The hypervisor (Virtual machine monitor) is the software layer which takes control over hardware resources and allocates them to VMs according to a given management policy.

Xen is one of the most popular free and open source baremetal hypervisors that are in use nowadays [3]. Xen's implementation follows the para-virtualization model [10]. In the latter, VMs' OS are modified to be aware of the fact that they are virtualized, thus reducing virtualization

overhead. In Xen, hardware components can be organized into two categories. Those which do not require drivers (e.g. memory and CPU) and the others which do (e.g. I/O devices). The logic for managing components of the first category is implemented at the hypervisor level. About components of the second category, things are little more tricky.

B. I/O virtualization in Xen

In Xen, the real driver for each I/O device resides within a particular VM called *privVM* (privileged VM). The *privVM* conducts I/O operations on behalf of VMs (called *unprivVM*) which run a fake driver called frontend, as illustrated in Fig. 1. The frontend communicates with the real driver via a backend driver (within the *privVM*) which allows multiplexing the real device. This I/O virtualization architecture is used by the majority of virtualization systems and known as the **the split driver model**. It enhances the reliability of the physical machine by isolating faults which occur within drivers. Thus, in the *privVM*, device drivers communicate with backend drivers (step (2) in Fig. 1 and these latter communicate with frontend drivers in *unprivVMs* (step (3)) allowing them to have a direct path to I/O devices. Finally, frontend drivers transmit the I/O request data to or from userspace applications (step (4)).

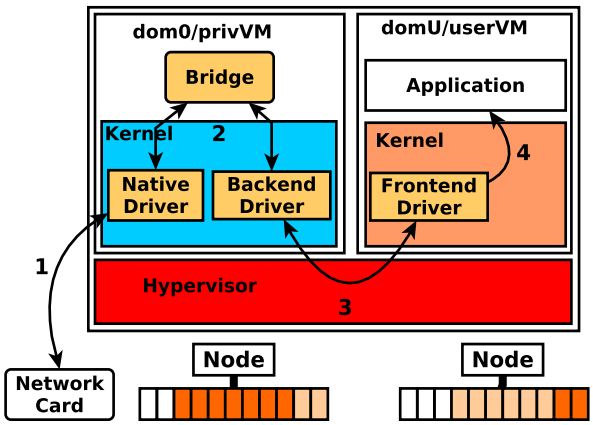

Fig. 1: Split driver model in Xen.

The communication mechanism (step (3) in Fig. 1) between a frontend driver and a backend driver in Xen is based on three elements: a ring mechanism, access grants on pages, and event channels. Let us consider a network communication from the backend driver to the frontend driver (knowing that the mechanism is similar in the opposite direction and also with a disk operation). When a network packet is transmitted to the backend driver, the latter builds a request containing all information on the received data and puts the request in the ring. Then, the backend driver notifies the frontend driver via event channels. The frontend driver reads the request and accesses the data. Given that the data is stored in pages of the *privVM*, the hypervisor has to give the access grants on these pages to the *unprivVM*. To counterbalance the transmission of these pages, the hypervisor gives access grants on some pages from the *unprivVM* to the *privVM*. This mechanism is known as **memory flipping** and allows a **zero-copy memory** communication between the backend driver and the frontend driver. During step (4) (Fig. 1), the guest OS copies the received data in the user space and frees the pages containing the data for further usage. The memory flipping mechanism provides a significant performance benefit for I/O applications but has a significant disadvantage for VMs on NUMA topologies.

C. NUMA in virtualization

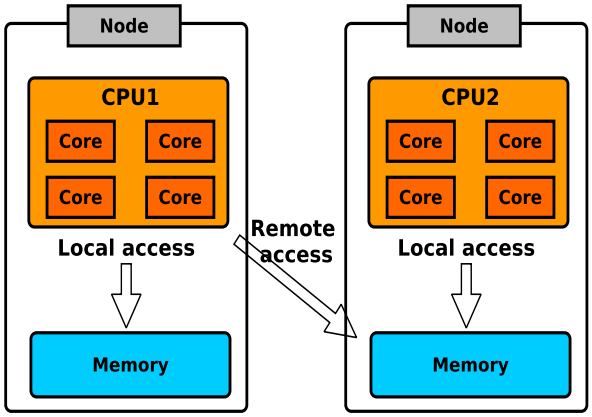

Fig. 2: A NUMA topology.

Today's machines have evolved to NUMA architectures. A NUMA architecture is an interconnection of several NUMA nodes that each contains a memory bank and several cores (see Fig. 2). To ensure good performance with such an architecture, the OSes must ensure that the applications always run on the node hosting their data to avoid remote memory access. Accessing local memory gives higher throughput and lower latency compared with remote memory access. The approach advocated by most of the hypervisors for handling NUMA architectures is the static approach with vNUMA. At the time of writing of the paper, vNUMA is supported by Xen, VMware and hyper-V.

With vNUMA, each VM is presented a static virtual NUMA (noted vNUMA) topology, which corresponds to the mapping of its allocated resources on NUMA nodes at boot time. The implementation of vNUMA consists in (for 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. Therefore, the hypervisor usually disables on VMs using vNUMA all mechanisms likely to modify their topology: VM migration and memory ballooning.

We noticed that memory flipping can be the cause of topology modifications for VMs as explained in Section I. The solution provided by Xen is not suitable because many memory copies lead to disastrous performance for I/O applications in *unprivVMs*. This statement is confirmed by an evaluation we carried out whose results are presented in Section III.

III. Motivations

As mentioned in Section II, the widespread configuration on a NUMA topology dedicates an entire node to the *privVM* to isolate it from *userVMs*. And memory flipping affects the NUMA topology of a *userVM* and severely impacts its performance. In this section, we performed a set of experiments with the following purposes:

1) to demonstrate that memory flipping actually affects the NUMA topology of a *userVM*;
2) to evaluate the impact on application performance;
3) to show that memory copy (which is the solution proposed by Xen) leads to significant performance degradation for I/O applications;

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

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

The obtained results are presented in Fig. 3 and Fig. 4. From left to right in Fig. 3, we have: the memory layouts of our *userVM*, respectively with memory flipping (a) and memory copy (b), and the performance of Big Bench (c). Fig. 4 shows the throughput of the Stream benchmark and the number of remote memory allocations during the executions of the Stream benchmark.

The first observation is on Fig 3(a) where we simply observe a progressive displacement of our *userVM* memory from its initial node to the *privVM*'s node with memory flipping. After 180 minutes of execution, 25% of the memory of our *userVM* has been relocated to the *privVM*'s node. This displacement is not observed with the memory copy (see Fig 3 (b)). However, in Fig 3(c), we note that the performance of Big Bench is significantly degraded with memory copy compared to memory flipping. This shows why memory copy cannot be a suitable solution to the problem we address. Fig. 4(left) shows the loss in performance for the Stream benchmark with memory flipping, about 13% between the two runs (first and second run). This is explained by the fact that an amount of our *userVM* memory which was relocated to the *privVM*'s node (consequently to the execution of Big Bench) is afterward used by the Stream benchmark. The solid lines in the histogram boxes represent the number of remote allocations during the executions of the Stream benchmark (the axis to which it refers to is on the right). There isn't any remote allocation with memory flipping during the first run, but this is not the case during the second run. This explains the performance degradation for the second run. With memory copy, the number of remote allocation remains nul. The main lessons we learn from these experiments are:

- memory flipping can effectively lead to NUMA topology changes for an *userVM*;

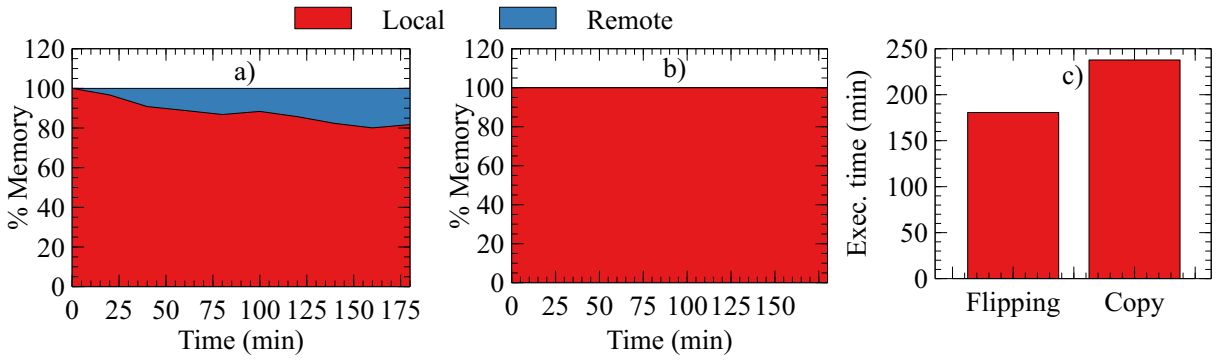

Fig. 3: unprivVM memory layout with memory flipping (a) and memory copy (b) - application performance for Big Bench (c).

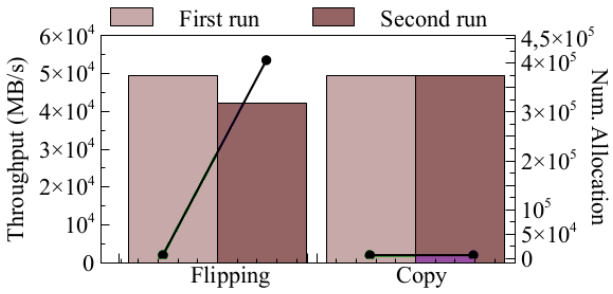

Fig. 4: Performance of the Stream benchmark: throughput (left axis / histogram boxes) and number of remote allocations (right axis / solid lines).

- memory copy is a solution to the problem, but causes significant performance degradation for I/O applications;
- the displaced memory of a *userVM* can later be used by applications, causing performance degradation.

In the rest of the article, we propose two effective solutions to solve this problem.

IV. Contributions

In this section, we propose two solutions to reduce the impact of memory flipping on a VM running on a NUMA architecture. The solutions we propose have been implemented within Xen hypervisor but can be easily integrated into other hypervisors implementing the vNUMA option.

A. Dedicated page pool for memory flipping

With the current implementation within Xen, after a memory flip, *unprivVM* memory pages which are relocated on the *privVM* node can be subsequently used by applications after they have been freed (step (4) of Fig. 1). Our first solution aims at limiting the number of memory pages that can be remote for an *unprivVM*, and also preventing these pages from being used by applications. At startup of an *unprivVM*, we create a pool of pages that are dedicated to memory flipping. Let's call $poolSize$ the size of the pool. These pages cannot be used by other applications in the *unprivVM*. After a memory flip, pages of *unprivVM* relocated on the *privVM* node are not freed anymore, but are rather added to the pool to replace the pages that have been yielded to the *privVM*, so that they will be used for subsequent flips and will be relocated on the *privVM* node. Thus, our solution limits the number of pages that can become remote for the *unprivVM* and also ensures that these remote pages cannot be used by any other application.

Fig. 5 illustrates the functioning of our solution in a simple scenario with the *privVM* running on node 1 and an *unprivVM* running on node 2. At step (1), all the pages of our *unprivVM* are on their initial node, and some pages are placed in the page pool for memory flipping. After a memory flip on a single page in step (2), a page of *unprivVM* is now on the *privVM* node. But this page cannot be used by any application in the *unprivVM* because it is part of the page pool dedicated to memory flipping. In step (4), after two successive memory flips, we can observe that only pages in the pool are used for flipping, thus limiting the number of pages from the *unprivVM* which can be remote. A key point of this solution is the $poolSize$ value. Two possible approaches can be used to define this value for a VM: the first is static while the second is dynamic.

Static value for $poolSize$. This approach consists in assigning a static value to $poolSize$ for a VM. This value is obtained by calibration. The latter is done by running several I/O applications in a VM and monitoring for each

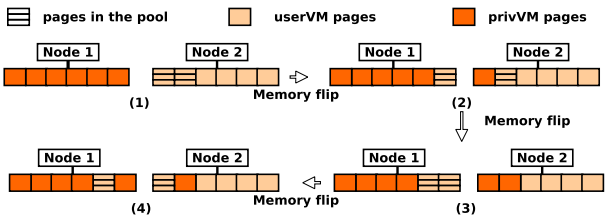

Fig. 5: Illustration of the dedicated page pool for memory flipping.

application the number of pages needed for the memory flipping. Then, we can use the highest number of needed pages as the value of $poolSize$. This approach has two big disadvantages: (1) calibration is a fastidious task which requires a lot of time and (2) with new application types appearing every day in the cloud, we cannot ensure that a single value of $pooSize$ will always be suitable, hence the need for a dynamic solution.

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

Algorithm 1 Dynamic poolSize estimation algorithm

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

B. Asynchronous memory migration

This solution repatriates the remote pages of an *unprivVM*, by migrating them asynchronously to their initial node. Initially, we let memory flipping behave normally. But at a given time, this solution starts a process which migrates the pages that have been relocated on the *privVM* node back to the *unprivVM* nodes. This allows the *unprivVM* to recover its initial topology and thus to avoid remote memory access for running applications. Fig. 6 presents a simple example of how this solution works on two nodes. In step (1), all the memory of the *unprivVM* is on the initial node. In step (2) and (3) we have two consecutive flips on memory pages. In step (4), our solution migrates the pages of the *unprivVM* to their initial node, which allows recovering the initial topology. Defining the frequency and the condition in which the memory migration process has to be started is the main problem with this solution. Two possible approaches can be used: the first is periodic and the second is based on the amount of *unprivVM* memory turned to remote.

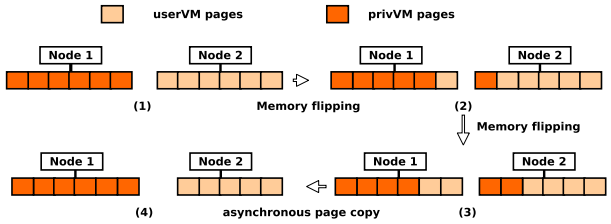

Fig. 6: Illustration of the asynchronous memory migration.

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

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

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

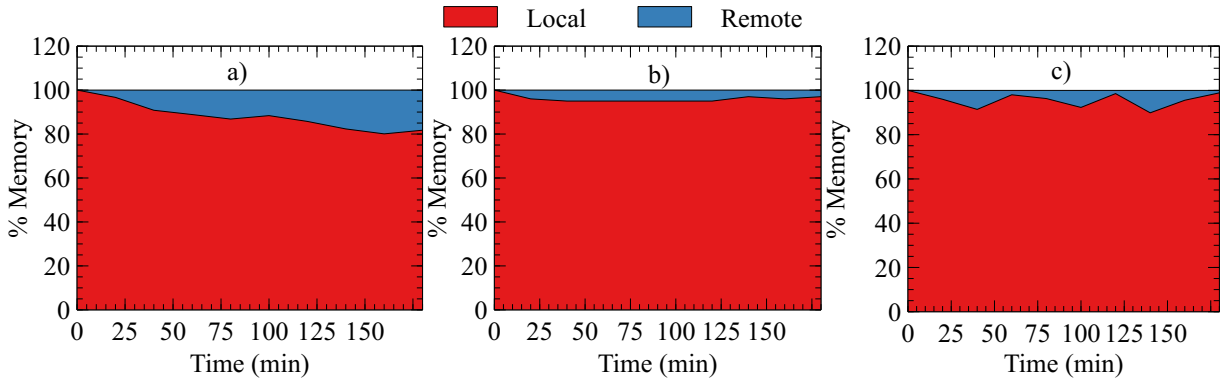

Fig. 7: unprivVM memory layout after running Big Bench: (a) for vanilla Xen, (b) for the page pool solution and (c) for the asynchronuous memory migration

V. Evaluation

This section presents the evaluation results of our solutions. We implemented prototypes of our solutions in Xen 4.7 [3].

A. Experimental setup and methodology

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

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

- *Big Bench.* [6] It is an open-source big data benchmark suite. Big Bench proposes several benchmark specifications to model five important application domains, including search engine, social networks, ecommerce, multimedia data analytics and bioinformatics. The metric used for this benchmark is the execution time.
- *LinkBench.* [2] LinkBench is a database benchmark developed to evaluate database performance for workloads similar to those of Facebook's production MySQL deployment. The metric used for this benchmark is the execution time.
- *Stream benchmark.* [8] Stream is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernels. The metric used for this benchmark is the memory bandwith.

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

For asynchronous memory migration, the results we present are those obtained with the second approach (the migration process is started when a given amount of remote pages is reached, see Secion IV-B). We observed that with the best value for $period_{flip}$ (900sec) and $remote\_memSize$ (500MB), the benchmarks have the same performance. Therefore, we decided to present only the results of the second approach with $remote\_memSize$ set to 500MB.

Methodology. Our evaluation aims to show that :

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

The experimental procedure we used is similar to the one in Section III. The procedure was repeated with vanilla Xen (with memory flipping), and with both of our solutions. Initially, we run the Stream or LinkBench benchmark and measure the performance (first run). Then, we execute Big Bench which is configured to perform I/O operations with another VM on a second server.

During the execution of Big Bench, we continuously monitor the memory layout of our *unprivVM*. Then, we rerun the Stream or LinkBench benchmark to obtain the new performance level (second run). This procedure is executed with vanilla Xen and both of our solutions.

B. Results analysis

Fig. 7 and Fig. 8 show the results of these experiments. Fig. 7 presents the memory layout of the *unprivVM* during Big Bench execution. We observe that with vanilla Xen, after 180 minutes of execution, about 25% of the memory of the *unprivVM* was moved to another node (Fig. 7(a)), thus modifying its NUMA topology. With both of our solutions, we observe that the amount of displaced memory is kept very small. This can be observed in Fig. 7(b) (page pool) and (c) (memory migration). Only a very little percentage (3%) of the memory is remote. Fig. 8 (a) and (b) present respectively the results for the Stream and LinkBench benchmarks, and the number of remote allocations. The solid lines in the histogram boxes of Fig. 8 represent the number of remote allocations during the executions (the axis to which it refers to is on the right). We can observe that the throughputs obtained with our solutions for the Stream benchmark (Fig. 8 (a)) are very close for the first and second run (the higher the better). This is not the case with vanilla Xen which has a performance degradation of about 13%. There isn't any remote allocation with memory flipping during the first run of the Stream benchmark, but this is not the case during the second run. This explains the performance degradation of the Stream benchmark. With our solutions, the number of remote allocation remains constant and nul. We observe a similar behavior with LinkBench in Fig. 8 (b).

VI. Related work

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

I/O and virtualization. The integration of I/O devices in virtualized environements has always been a subject of primary importance. Many research works investigated the improvement of I/O operations' performance. We can distinguish three main categories of work according to the implementation level: (1) hardware level works which focus on hardware modification to improve the I/O performance [1], (2) hypervisor level works which propose new scheduling approaches to reduce the latency of I/O requests [12], [13] and finally, (3) guest OS-level works which focus on the optimization the software layer for the I/O processing in the *userVM* and the *privVM* [11]. Works on memory flipping and memory copy techniques are in this third category. The introduction of para-virtualization with Xen (which is the flagship), pushed the hypervisor designers to propose more efficient I/O architecture, hence leading to the split driver model. The latter allows isolation for fault tolerance while offering interesting performance compared to full virtualisation. The first implementation of the split driver model within Xen was based on the memory copy. But the performance of applications with this approach was very low and had to be improved. Therefore, Xen introduced memory flipping which ensures zero memory copy during the processing of I/O operations. Several works have studied the split driver model and the frontend/backend communication mechanism [4], [11]. The most interesting of these works proposes to establish a shared memory region between all the VMs for implementing frontend/backend communication [4]. This approach seems interesting because it provides much better performance for I/O applications compared to memory flipping and also, it avoids the problem that we solve in this article. However, one important limitation of this approach, which is significant considering that we are in the context of the Cloud, is security. Indeed, establishing a shared memory region between all the VMs breaks VM isolation and opens a gateway to many exploits from malicious hackers.

NUMA and virtualization. NUMA management is ubiquitous in both virtualized and non-virtualized environments. Since the appearance of NUMA architecture, a plethora of scientific works have focused on handling NUMA in non virtualized environments [5], [7]. Generally, studies on NUMA in virtualized environments are simple translations of solutions in non-virtualized environments [9]. The main question that researchers had to answer is the place where NUMA specificities should be handled: in the hypervisor or in the guest OS. Most hypervisors (Xen, VMWare and Hyper-V) have adopted a simple solution called vNUMA. The latter consists in presenting the NUMA topology of the VM to the guest OS and preserving this topology (by disabling operations that would modify it). This solution is straightforward because, it allows the guest OS to benefit from all the already implemented kernel optimisations for NUMA. However, some studies demonstrated that this approach is not suitable for some applications which would need a dynamic NUMA approach [9].

Position of our work. To the best of our knowledge, we are the first work which addresses the problem of the compliance between vNUMA and memory flipping,

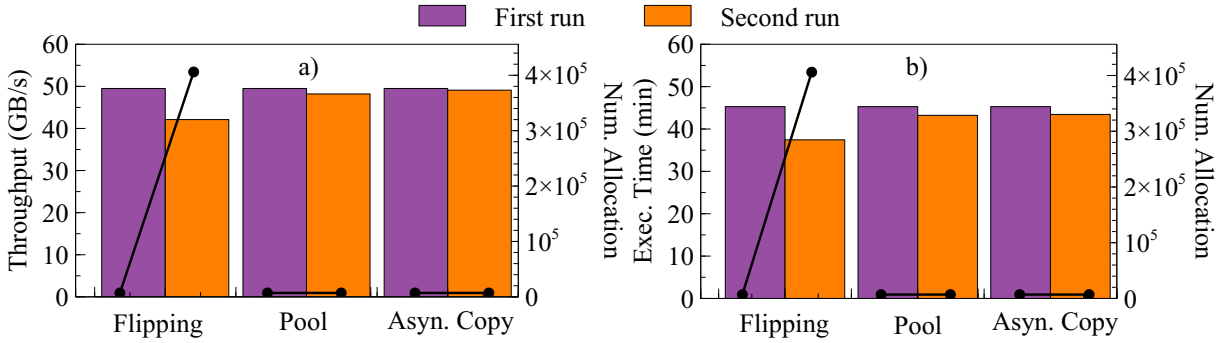

Fig. 8: (a) Stream and (b) LinkBench benchmark results.

which are techniques used by almost all hypervisors. The only existing solution is that of Xen which relies on memory copy, thus degrading I/O performance in VMs. Our solutions allow to keep the benefits of locality in NUMA architectures while also keeping good performance for I/O applications.

VII. Conclusion

We observed that after a significant number of memory flips, a notable part of the memory of *unprivVMs* is moved to the node of the *privVM*, thus modifying their NUMA topology and forcing these VMs to make remote memory access. This situation has a negative impact on application performance in the VMs. In this article, we proposed two solutions which attempt to enforce a static NUMA topology for VMs despite memory flipping. We have implemented and evaluated our solutions in the Xen hypervisor. The evaluation shows that our solutions allow to get close to a static topology for VMs, thus limiting remote memory access and providing better performance than vanilla Xen for I/O applications.

VIII. Acknowledgment

This work was supported by the HYDDA Project (BPI Grant) and the IDEX IRS (COMUE UGA grant).

References

[1] Sr-iov. https://pcisig.com/specifications/iov/. Visited on October 2017.

[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.

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

[4] R. C. Chiang, H. H. Huang, T. Wood, C. Liu, and O. Spatscheck. Iorchestra: Supporting high-performance data-intensive applications in the cloud via collaborative virtualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 45:1–45:12, New York, NY, USA, 2015. ACM.

[5] 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 Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[6] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013. ACM.

[7] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 277–289, Berkeley, CA, USA, 2015. USENIX Association.

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

[9] G. Voron, G. Thomas, V. Quéma, and P. Sens. An interface to implement numa policies in the xen hypervisor. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 453–467, New York, NY, USA, 2017. ACM.

[10] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, Dec. 2002.

[11] 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 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 243–254, Berkeley, CA, USA, 2013. USENIX Association.

[12] 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 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 3–14, New York, NY, USA, 2012. ACM.

[13] L. Zeng, Y. Wang, W. Shi, and D. Feng. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *2013 International Conference on Cloud Computing and Big Data*, pages 267–274, Dec 2013.