Closer: A new design principle for the privileged virtual machine OS

Djob Mvondo
LIG
Grenoble Alpes University
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
Grenoble Alpes University
Grenoble, France
noel.depalma@univ-grenoble-alpes.fr

***Abstract*—In most of today's virtualized systems (e.g., Xen), the hypervisor relies on a privileged virtual machine (pVM). The pVM accomplishes work both for the hypervisor (e.g., VM life cycle management) and for client VMs (I/O management). Usually, the pVM is based on a standard OS (Linux). This is source of performance unpredictability, low performance, resource waste, and vulnerabilities. This paper presents *Closer*, a principle for designing a suitable OS for the pVM. *Closer* consists in respectively scheduling and allocating pVM's tasks and memory as close to the involved client VM as possible. By revisiting Linux and Xen hypervisor, we present a functioning implementation of *Closer*. The evaluation results of our implementation show that *Closer* outperforms standard implementations.**

***Index Terms*—Virtualization, Privileged VM, NUMA, Para-Virtualization**

I. Introduction

The last decade has seen the widespread of virtualized environments, especially for the management of cloud computing infrastructures which implement the Infrastructure as a Service (IaaS) model. In a virtualized system, a low level kernel called the hypervisor runs directly atop the hardware. The hypervior's main goal is to provide the illusion of several virtual hardware to guest OSes (called virtual machines - VMs). In order to keep its trusted computing base as smaller as possible, some of its functionalities (such as VM monitoring and life-cycle management tools) are provided by a privileged virtual machine (hereafter pVM), see Fig. 1.

Para-Virtualization (PV), introduced by Disco [1] and Xen [2], is a very popular virtualization approach which consists in making the guest OS aware of the fact that it runs in a virtualized environment. This approach has been proven to reduce virtualization overhead [3] and also mitigate the limitation of some virtualization hardware features (e.g. the inability to live migrate a VM with SR-IOV [4]–[6]). In this approach, the pVM plays a critical role because it hosts some device drivers (e.g. I/O) used by unprivileged VMs (*uVMs*), making the pVM performing a lot of work on behalf of uVMs. In this paper, we are interested in virtualization systems which rely on such a pVM.

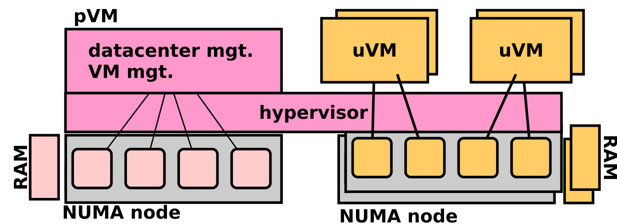

Fig. 1. Overall architecture of a virtualized system

The main issue with current pVMs is that they rely on standard OSes (e.g., Linux) whose resource manager does not consider pVM's particularities. The most important characteristic of the pVM is **the correlation between pVM's tasks and uVM's activities**. A significant part of pVM tasks are correlated with uVM's activities: (1) in terms of quantity, as the amount of resources required by the pVM depends on uVM's activities (especially I/O operations), and (2) in terms of location in a NUMA architecture, as resources (e.g. memory) allocated for one pVM's task are likely to be used by correlated uVM's activities.

A standard OS (running in a VM) manages resources for its applications and itself, but is not aware of resources managed in other VMs running on the same host. Therefore, the previous correlation between pVM and uVMs is not considered in a standard OS, thus leading to resource waste, low performance on NUMA architectures, performance unpredictability, and vulnerability to DoS attacks [7]–[9]. To take into account this correlation, the pVM should respect three rules:

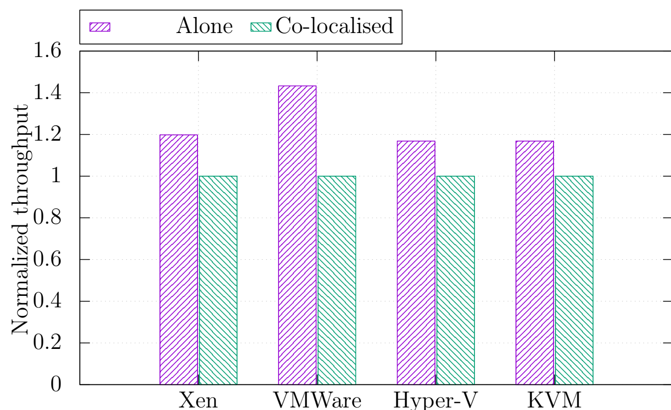

Fig. 2. In all existing hypervisors, the correlation of pVM's tasks with uVM's activities is not taken into account. This leads to several issues such as performance unpredictability, illustrated in this figure. Normalized performance of a website benchmark (wordpress) executed in two situations: alone and colocated with several other I/O intensive uVMs (normalized over the co-located situation). A predictable pVM management would have led to the same performance level regardless the colocated uVMs. The hypervisors versions are respectively Xen 4.9.1, VMWare ESXi 6.5, Hyper-V 2016 and Qemu-KVM 2.10.1

- **on-demand**. Resource allocation to the pVM should not be static. Resources should be allocated on demand according to uVMs activities. Without this rule, resources would be wasted if over-provisioned or they would be lacking, leading to performance unpredictability.
- **pay-per-use**. The resources allocated to a pVM task which is correlated with a uVM activity should be charged to the concerned uVM. Without this rule, the pVM would be vulnerable to DoS attacks from a VM executing on the same host, which could use most of the resources from the pVM.
- **locality**. The resources allocated to a pVM task which is correlated with a uVM activity should be located as close as possible (i.e. on the same NUMA node) to the uVM activity. This rule allows reducing remote memory accesses in the NUMA architecture, improving the uVM performance.

These rules are not currently respected by the pVM of the most popular virtualization systems.

In order to assess the significance of this issue, we evaluated the performance of a website (provided with wordpress [10]) executed within a VM in two situations. First, the VM runs alone on the machine. Second, the VM shares the machine with other VMs which perform intensive I/0 operations. We performed these experiments using popular hypervisors. In Fig. 2, we can observe that for all hypervisors, the performance of wordpress is unpredictable. We suspect (and we later demonstrate it with Xen) that this is caused by the saturation of the pVM, which size is fixed, thus not correlated with uVM's activities. In fact, by over-estimating the number od CPUs assigned to the pVM, we observed no performance variation.

In this paper, we propose *Closer*, a principle for implementing a suitable OS for the pVM, *Closer* promotes the proximity (to uVMs) and the utilization of uVM's resources. Following *Closer*, we propose a general architecture for the pVM. We revisit Linux thanks to this architecture, thus providing a ready to use OS dedicated to the pVM. We demonstrate the effectiveness using Xen, a popular open source virtualization system (used by Amazon). The final contribution of this paper is the instensive evaluation of our implementation usign both micro- and macro-benchamrks. The evaluation results show :

- No resource waste from the pVM, as the resources are allocated on-demand.
- Improvement of both VM life-cycle management tasks (creation, destruction, migration, etc.) and user's application performance in comparison with vanilla Linux (as the pVM OS) and Xen (as the hypervisor). We improve both VM destruction and migration time (by up to 33%). Regarding user's application performance, we improve I/O intensive workloads (up to 36.50% for packet reception latency, 42.33% for packet emission latency, and 22% for disk operations).
- Pay-per-use and performance predictability enforcement, as pVM used resources are charged to the uVMs it works for.

The rest of the paper is organized as follows. Section II introduces key characteristics of virtualization systems where our works take place. Section III presents the motivations and an assessment of the problem. Section IV presents our contributions while Section V presents evaluation results. A review of the related work is presented in Section VI. The conclusion is drawn in Section VII.

II. Background

Fig. 1 illustrates the general architecture of a virtualized environment. The hypervisor is the system layer which enforces resource sharing between virtual machines. For simplicity, maintainability and security purposes, a particular VM is used as an extension of the hypervisor. This VM is called the privileged VM (*pVM*) while other VMs are called unpriviledged VMs (*uVMs*). The pVM embeds uVM life-cycle administration tools (e.g., `libxl` in Xen, `parent partition` in Hyper-V) and data center administration applications (e.g *novaCompute* in OpenStack). In addition to these tasks, the pVM is also used in most deployments as a proxy for sharing and accessing I/O devices, see Fig. 3. In this case, the pVM embeds the driver enabling access to the hardware device, and a proxy (called backend) which relays incoming (from the driver to a uVM) or outgoing (from a uVM to the driver) requests. Each uVM includes a fake driver (called frontend) allowing to send/receive requests to/from the uVM's backend. Therefore, each I/O operation performed by a uVM involves significant computations in the pVM. This architecture for I/O management is known as the **split-driver** model. It is widely used in datacenters for its very good performance and its high flexibility compared to hardware-supported (HVM) solutions based on Single Root I/O Virtualization (SR-IOV) [11]. Indeed, SR-IOV allows a single

device to be shared by multiple VMs while bypassing both the hypervisor and the pVM (the VM sends/receives packets directly to/from the device). However, SR-IOV is not widely adopted in datacenters as it comes with a strong limitation: VM migration should be disable [4]–[6], thus forbidding VM consolidation in the datacenter (which is critical for energy saving).

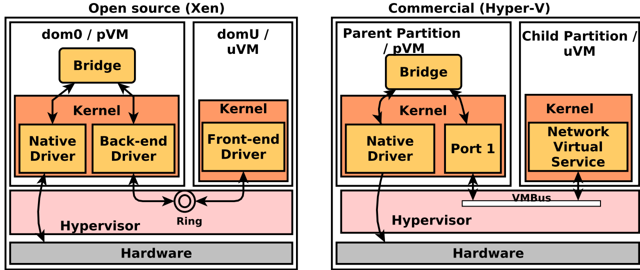

Fig. 3. Both open source (Xen) and commercial (Hyper-V) hypervisors implement the split-driver model. In the latter, the pVM provides access to I/O devices on behalf of uVMs.

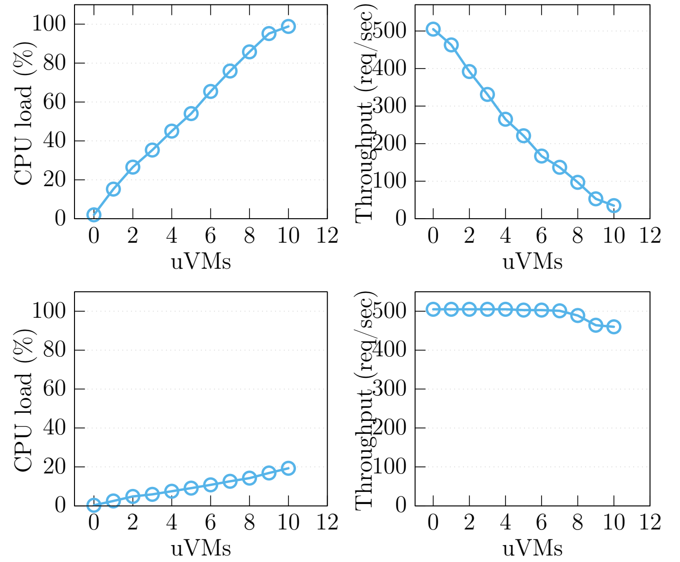

Fig. 4. A static resource allocation to the pVM can lead to its saturation, thus to performance unpredictability for user applications: (left) pVM's CPU load, (right) performance of the uVM which executes a (Wordpress) workload. The pVM has 2 CPUs for the top results and 12 CPUs for the bottom.

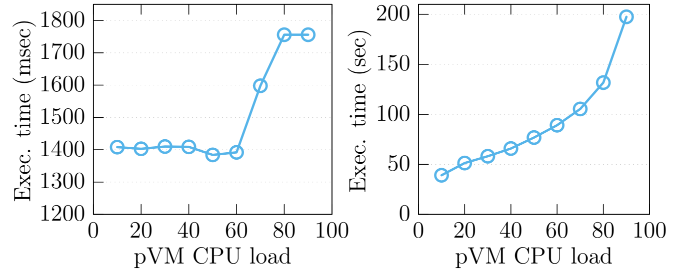

Fig. 5. A static resource allocation to the pVM may lead to its saturation, thus to variable uVM (left) creation and (right) migration time.

III. Motivations

Almost all previous research works have focused on resource allocation to uVMs since they host user applications. This paper shows that resource management for the pVM is a tricky task, and that it can have a significant impact on both administrative tasks and user applications. We are investigating this issue especially in NUMA architectures which are commonly used in today's datacenters.

A pVM resource management strategy must consider the three fundamental questions:

- Q_1: how much resources (number of CPUs, amount of memory) should be allocated to the pVM?
- Q_2: how to organize such an allocation (in terms of location) in a NUMA architecture?
- Q_3: who (provider or user) these resources should be charged to?

The next sections discuss the current answers to those questions, and the related consequences.

A. (Q_1) pVM sizing

Building on our experience (several collaboration with several datacenter operators[1]), we observed that static configuration is the commonly used strategy for pVM sizing, which means that the pVM is allocated a fixed amount of resources at startup. Moreover, virtualization system providers do not provide any recommendation regarding this issue. The only recommendation we found comes from Oracle [12] which proposes only a memory sizing strategy for the pVM based on the following formula: $pVM_mem = 502 + int(physical_mem \times 0.0205)$. More generally, there isn't any defined method to estimate the amount of resources required by the pVM to perform correctly.

The resources required by the pVM are not constant as they depend on uVM activities, therefore a static allocation cannot be the solution. In fact, the tasks executed by the pVM can be organized in two categories: tasks related to the management of the datacenter and tasks related to uVM I/O operations. The amount of resources required by tasks from the second category is fickle as it depends on uVM activities. A static allocation can lead to two situations, either the pVM's resources are insufficient or overbooked. These two situations can be harmful for both the cloud provider and user applications. We focus on the consequences of an insufficient resource provisioning to the pVM because it is obvious that over-provisioning causes resource waste as shown by many researchers [7]. The experimental environment used in this section is detailed in Section V-A.

A lack of resource in the pVM can make both applications executed in uVMs and administrative services executed in the

[1]One of our datacenter operator partner is in charge of up to 308 virtualized clusters around the world

pVM inefficient and unpredictable. The latter is known as one of the main issues in the cloud [8], [13], [14].

Impact on user's applications

Fig. 4 top right shows the performance of a uVM which hosts a web application (wordpress) on a physical machine where we vary the number of colocated uVMs which execute I/O intensive workloads. In this experiment, the pVM is allocated two CPUs while each uVM is allocated one CPU. To prevent any contention (e.g. QPI link contention), all CPUs (from the pVM or uVMs) are allocated on the same NUMA socket. Fig. 4 top left presents the pVM CPU load during each experiment. The first observation is that the pVM load varies according to uVMs activities. This is due to the fact that the pVM embeds both the backends and the drivers responsible for accessing I/O devices (see Fig. 3). The second observation is that the web application's performance decreases when the pVM lacks CPU resources. Therefore, performance predictability is compromised.

One may wonder whether this unpredictability is effectively caused by lack of computation power for the pVM. Since all uVMs execute I/O intensive workloads, the I/O hardware could be the bottleneck. To clarify this point, we ran the same experiment with 12 CPUs for the pVM. The results on Fig. 4 bottom show that with enough resources, the performance of the tested application remains almost constant, which proves that resources allocated to the pVM are the bottleneck.

Impact on management tasks

The pVM hosts VM management operations, the most important ones being: VM creation, destruction and migration. The saturation of the pVM can lead to execution time variation for these operations since they require a significant amount of resources. Fig. 5 left and right respectively show VM creation and migration times according to the pVM load. We observe that the pVM load has a significant impact on these execution times. This situation may dramatically influence cloud services such as auto-scaling [8].

B. (Q_2) pVM location

On a NUMA machine, the location of the resources allocated to the pVM may significantly influence uVMs performance. The commonly used strategy is to locate all pVM resources on a dedicated NUMA socket, not used by any uVM. This section shows that running the pVM close to uVMs may improve the performance of the latter.

I/O intensive applications' improvement

We executed a web application in a uVM whose entire CPU and memory resources were located on the same NUMA socket as the pVM. Then, we varied the location of the pVM resources. We observed that the best performance is obtained when the pVM and the uVM share the same NUMA socket (521 req/sec and 8.621 ms latency if colocated vs 491 req/sec and 13.431 ms latency if not). This is because colocation on the same socket prevents remote memory accesses (to fetch I/O packets) from both pVM and uVM tasks.

VM migration and destruction improvement

We also observed that running the pVM close to uVMs may improve some management tasks such as live migration. For instance, we observed in our testbed that the migration of a 2GB RAM uVM can be improved by about 34.15% if the migration process running in the pVM is scheduled on the same NUMA socket which hosts the migrated uVM's memory. We made a similar observation for uVM destruction tasks: the scrubbing step (memory zeroing) is much faster when the migration process runs close to the NUMA socket which hosts the uVM's memory.

C. (Q_3) pVM resource charging

The commonly used strategy is to leave the provider support the entire pVM resources, which includes the resources used on behalf of uVMs for performing I/O operations. [9] showed that this is a vulnerability, which could lead to deny of service attacks.

IV. Contributions

A. Design principle

We introduce *Closer* (promotes proximity to uVMs), a design principle for implementing a pVM which solves the previously identified issues.

The issues presented in Section III can be summarized by one word : **correlation**. If the pVM relies on a standard OS, its resource management does not consider the correlation between pVM tasks and uVMs activities. However, a significant part of pVM tasks are correlated with uVMs activities : (1) in terms of quantity, as the amount of resources required by the pVM depends on uVMs activities (especially I/O operations), and (2) in terms of location in a NUMA architecture, as resources allocated for one pVM task (e.g. memory) are likely to be used by correlated uVM activities.

To take into account this correlation, our *Closer* principle influences the design of the pVM with the three following rules:

- **on-demand**. Resource allocation to the pVM should not be static. Resources should be allocated on demand according to uVMs activities. Without this rule, resources would be wasted if over-provisioned or they would be lacking, leading to performance unpredictability.
- **pay-per-use**. The resources allocated to a pVM's task which is correlated with a uVM activity should be charged to the concerned uVM. Without this rule, the *pVM* would be vulnerable to DoS attacks from a VM executing on the same host, which could use most of the resources from the pVM.
- **locality**. The resources allocated to a pVM's task which is correlated with a uVM activity should be located as close as possible (i.e. on the same NUMA socket) to the uVM activity. This rule allows reducing remote memory accesses in the NUMA architecture.

To enforce the *Closer* principle, we introduce an architecture where the pVM is organized in two containers (see Fig. 6): a *Main Container* (MC) and a *Secondary Container* (SC). Each

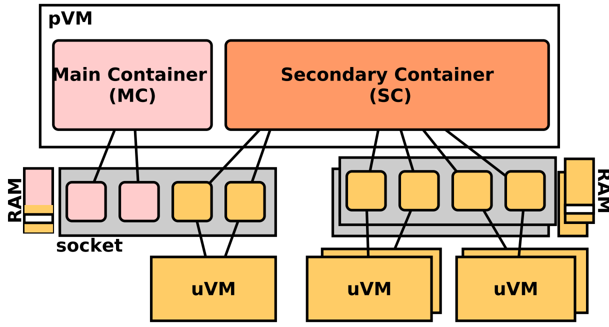

Fig. 6. The new pVM architecture.

container is associated with a specific resource management policy, which controls pVM's resource mapping on physical resources. We implemented this architecture by revisiting Linux instead of building a new pVM from scratch. By this way, we propose a ready to use pVM.

The MC is intended to host tasks (i.e. processes) whose resource consumption is constant (i.e. do not depend on uVM activities). Other tasks which depend on the activity of a uVM are hosted in the SC. More precisely, pVM tasks are organized into four groups: (T_1) OS basic tasks (Linux in our case), (T_2) tasks belonging to the datacenter administration framework (e.g. *novaCompute* in OpenStack), (T_3) VM management tasks (create, destroy, migrate, etc.) and (T_4) I/O management tasks (drivers and backends, see Fig. 3). Tasks from T_1 and T_2, and almost all tasks from T_3 (except VM destruction and migration tasks) have a constant resource consumption and are executed in the MC. All other tasks use resources according to uVMs activities and they are executed in the SC.

B. Resource management

Description (see Fig. 6)
At physical machine startup, the pVM is configured with as many vCPUs as available physical processors (called cores). Each vCPU is pinned on a core (one per core). A subset of vCPUs (therefore of cores) of the pVM is associated with the MC (i.e. used for MC tasks). The rest of vCPUs are linked to the SC and their associated cores are shared with uVMs. Therefore, when a core is reserved for a uVM (the core is the allocation unit), two vCPUs are pinned on that core: the uVM's vCPU and the SC's vCPU associated with that core. This allows the SC to execute the tasks correlated with the uVM on its reserved core (therefore to charge the used resources to the uVM), following the **pay-per-use** and **locality** rules. Regarding the **on-demand** rule, the MC is allocated a fixed amount of resources (vCPU and memory) according to its constant load and the SC is granted a variable amount of resources according to tasks scheduled on its vCPUs.

Resource management for the main container
The resources allocated to the MC must be on provider fee, as they are used for executing datacenter administrative tasks (e.g. monitoring). These resources are constant and can be estimated as they only depend on the datacenter administration system (OpenStack, OpenNebula, CloudStack, etc.). Neither the number of VMs nor VMs' activities have an impact on MC resource consumption. Therefore, we use a static allocation for MC resources at physical machine startup and these resources are located on a reduced number of processor sockets. Through calibration, we can estimate the resource to be allocated to the MC. The evaluation section (Section V) provides such estimations for the most popular cloud administration systems.

Resource management for the secondary container
The SC includes as many vCPUs as available cores on the physical machine, excluding cores allocated to the MC. At physical machine startup, the SC hosts I/O tasks from the split-driver model (see Fig. 3) for uVMs. Even if the I/O tasks are not active at this stage, they require memory for initialization. This initial memory (noted `SCInitialMem`[2]) is very small and assumed by the provider. It will be also used for uVM destruction and migration tasks. Algorithm 1 synthesizes the task scheduling policy in the SC. When a uVM is at the origin of a task (e.g. for an I/O operation), one of its vCPU is scheduled-out and its associated core is allocated to the SC's vCPU mapped to that core, in order to execute this task. The following sections detail the implementation of this algorithm for I/O, destruction and migration tasks.

```
Input  : T: an I/O task or VM administration task that
         should run within pVM
1 targetuVM=Identification of the target uVM
2 targetCPU=Pick a CPU which runs targetuVM
3 Schedule T on the pVM's vCPU which is pinned on
  targetCPU
```
Algorithm 1: Task scheduling in the SC.

C. I/O scheduling in the SC

pVM's tasks which are executing I/O operations on the account of uVMs are twofold: backends and drivers. The challenge is to identify from the pVM the uVM responsible for each task, in order to use one of its allocated processors for scheduling this task, and to allocate I/O memory buffers as close to that processor as possible. Given that the split-driver structure is both used for network and disk I/O, we describe in the following our solution for networking tasks which are of two types: packet reception and emission from a uVM.

Packet reception
For illustration, let us consider the Xen implementation (see Fig. 7), although the description below could be applied to other hypervisors which rely on the split-driver model. When

[2]The amount of `SCInitialMem` depends on the number of backend tasks which is bound by the number of vCPU in the SC. We estimated that `SCInitialMem` accounts for about 10MB per backend instance in Xen as an example.

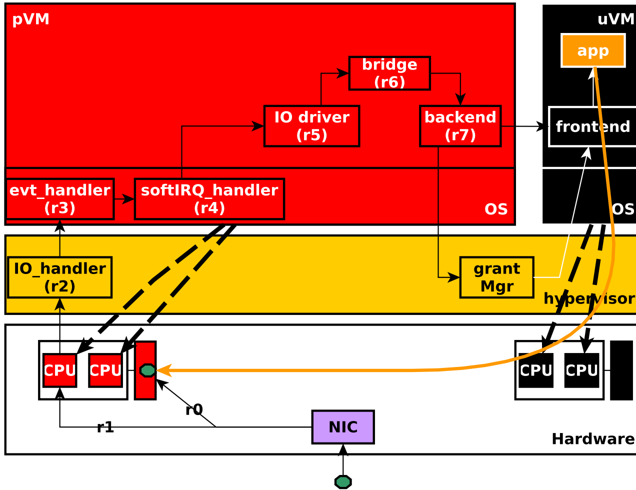

Fig. 7. Packet reception workflow.

the network adapter receives a packet, (r_0) it places the packet in a queue which was initially allocated in main memory and then triggers an interrupt. (r_1) This interrupt is transmitted to one of the processors of the physical machine by the IOAPIC. (r_2) The interrupt handler which lies in the hypervisor notifies to the pVM the presence of a packet as follows. A vCPU from the pVM (generally vCPU 0) is notified (thanks to the event channel mechanism) and is responsible for reacting to this event. The hypervisor then boosts (prioritizes) this vCPU in its scheduler. When the vCPU is scheduled, it detects the presence of an event and executes the event handler (r_3). This handler generates a softIRQ on one of the vCPU from the pVM. The handler of this softIRQ (r_4) triggers the treatment of the packet which will flow up in the network protocols. There are actually two ways of treatment: the traditional treatment which works on a per packet basis (we call it OAPI for old API) and a new one (that we call NAPI for new API) which groups message handling in order to reduce the number of interrupts. (r_5) In both cases, the packet has to be copied from the queue to a `skbuff` structure (via the `copybreak` primitive) and the network adapter can then be notified that the packet was well received. The packet is then forwarded to the protocol stack according to the protocol identified in the packet header. In a virtualized system, the destination of the packet is the *bridge*. (r_6) The bridge identifies from the packet header (which includes a target MAC address) the destination backend. The packet then flows down in the protocol stack to the backend. (r_7) On reception, the backend shares with the destination uVM the memory pages which include the packet (this is called *page flipping*) and sends a signal to that uVM. The intervention of the pVM stops here and the packet continues its path in the uVM (starting from the frontend).

In order to implement our *Closer* principle, the general orientation is to force the execution of all r_i steps on one of the processors of the target uVM and to allocate the buffer for the packet on the same socket as that processor.

After step r_0, the incoming packet has been inserted in a queue in main memory and an interrupt triggered on one processor (generally processor 0). The main issue is then to execute the other steps on a processor of the target uVM, while the target uVM is known only at the level of the bridge (step r_6). Regarding step r_1, we rely on *IRQbalance* [15] to balance interrupts between SC's processors (associated with uVMs vCPUs). It does not guarantee that interrupts will be handled by a processor from the target uVM, but it ensures that the MC is not charged for these interrupt handlings, which will be uniformly executed by uVMs processors. This is unfair for uVMs which execute less I/O operations, but the unfairness is limited to step r_1 and mitigated as follows. In the hypervisor, we monitor the I/O activity of each uVM[3] and the notification of a vCPU from the SC (step r_2) is done in proportion to the I/O activity of each uVM (the uVM with the highest I/O activity will receive more notifications on its processors, i.e. SC's vCPUs pinned on these processors will be more solicited). This solution (called hyperLB[4]) is fair, but not precise and inadequate regarding memory management, as the memory hosting the `skbuff` buffer will be allocated on the local socket (of the selected processor from the most I/O loaded uVM) which could be different from the socket of the final target processor. To prevent such a situation, we perform the identification of the target backend (and therefore of the target uVM) in step r_4 (it was previously performed in step r_6 in the bridge) and force the execution of the following steps on one of the processors of that uVM. This solution performs well in the OAPI case where each packet reception generates an interrupt and the execution of all the steps for each packet. In the NAPI case, a single interrupt can be generated for the reception of a group of packets (with different target uVMs), whose treatment relies on a poll mechanism. We modified the poll function within the driver (we did it for *e1000*, this is the only non generic code) in order to identify the target uVM for each polled packet. In the rest of the document, we call NAPI-1 the per group implementation while NAPI-n refers to the per packet implementation.

Packet emission

The out-going path is the reverse of the receive path presented in Fig. 7. Applying our *Closer* principle is in this case straightforward since the uVM responsible for the I/O activity is known from the beginning step. We simply enforce the hypervisor to notify the vCPU of the SC associated with a processor of the uVM. Then, we modified the pVM's scheduler for the following steps to be executed on the same processor. The same implementation is used for disk operations.

D. uVM destruction and migration scheduling in the SC

uVM destruction and migration are administration tasks which are also executed in the SC. Logically, resources consumed by these tasks should be charged to the provider.

[3]The monitoring is implemented at the level of the interface between the pVM backend and the uVM frontend.

[4]LB stands for load balancing.

In our solution, this is effectively the case for memory (with the allocation of `SCInitialMem` described above), but not for CPU which is charged to the concerned uVM. This is not a problem as the goal is here to remove the uVM from the host, and it has a main advantage: the proximity with the memory of the uVM.

uVM destruction
The most expensive step in the destruction process of a uVM is memory scrubbing [8]. This step can be accelerated if the scrubbing process executes close to the uVM memory (following the **locality** rule). However, the current implementation consists of a unique process which performs that task from any free vCPU from the pVM. Our solution is as follows. Let S_i, $i \leq n$ be the sockets where the uVM has at least one vCPU. Let S'_j, $j \leq m$ be the sockets which host memory from the uVM. For each S_i, a scrubbing task is started on a vCPU from the SC, the processor associated with this vCPU being local to S_i and shared with a vCPU of the uVM. This task scrubs memory locally. The uVM memory hosted in a S'_j which is not in S_i is scrubbed by tasks executing on other sockets (this remote scrubbing is balanced between these tasks).

uVM live migration
uVM live migration mainly consists in transferring memory to the destination machine. The current implementation consists of a unique process which performs the task from any free vCPU from the pVM. The transfer begins with cold pages. For hot pages, the process progressively transfers pages which were not modified between the previous transfer and the current one. An already transferred page which was modified is transferred again. When the number of modified pages becomes low or the number of iterations equals five, the uVM is stopped and the remaining pages are transferred. Then, the uVM can resume its execution on the distant machine. We implemented a distributed version of this algorithm which runs one transfer process per socket hosting the uVM's memory (similarly to the destruction solution), thus enhancing **locality**. These processes are scheduled on SC's vCPUs associated with free processors if available. Otherwise, the processors allocated to the uVM are used.

V. Evaluations

This section presents the evaluation results of our revisited Linux and Xen prototype.

A. Experimental setup and methodology

Servers. We used two Dell servers having 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; the SATA disk is also equidistant to the sockets. We used Xen 4.7 and both the pVM and uVMs run Ubuntu Server 16.04 with Linux kernel 4.10.3. Unless otherwise specified, each uVM is configured with four vCPUs, 4GB memory and 20GB disk.

TABLE I
THE BENCHMARKS WE USED FOR EVALUATIONS.

Type	Name	Description
Micro	*Memory*	a memory intensive application - live migration evaluation. It builds a linked list and performs random memory access as in [16]. It has been written for the purpose of this article.
	Netperf [17]	Sends UDP messages - network evaluation. The performance metric is the average request latency.
	dd	formatting a disk file - disk evaluation. It is configured in write-through mode. The performance metric is the execution time.
Macro	*Wordpress* [10]	website builder - network evaluation. We used the 4.8.2 version. The performance metric is the average request latency.
	Kernbench [18]	runs a kernel compilation process - disk evaluation. We compiled Linux kernel 4.13.3. Data caching is disabled. The performance metric is the execution time.
	Magento [19]	eCommerce platform builder - both network and disk evaluation. We used the 2.2.0 version. The performance metric is the average request latency.

Benchmarks. We used both micro- and macro-benchmarks, respectively for analyzing the internal functioning of our solutions and for evaluating how real-life applications are impacted. Table I presents these benchmarks.
Methodology. Recall that the ultimate goal of our proposed pVM architecture is to respect the three rules of *Closer*, presented in Section IV-A: on-demand resource allocation (to avoid resource waste and enforce predictability), locality (to improve performance), and pay-per-use (to prevent DoS attacks). As a first step, we separately demonstrate the effectiveness of our architecture for each principle. Notice that the evaluation of a given principle may not include all contributions. Therefore, a final experiment is realized with all the contributions enabled. For each experiment, we compare our solution with the common pVM's resource allocation strategy (referred to as Vanilla pVM, *Vanilla* for short). In the latter, a set of resources, which are located at the same place, are dedicated to the pVM. Otherwise specified, we dedicate a NUMA socket and its entire memory node to the pVM, corresponding to an oversized allocation.

B. Resource allocation to the MC

In our solution, MC's resources are statically estimated by the provider. We estimated MC's resources for the majority of cloud management systems namely: OpenStack Ocatax [20], OpenNebula 5.2.1 [21] and Eucalyptus 4.4.1 [22]. For each system, we relied on the default configuration and measured its resource consumption. The results, based on our testbed, are presented in Table II. We can see that very few resources are needed for performing all MC's tasks. Our experiments also confirmed that MC's needs do not depend on the IaaS size.

C. Locality benefits

We evaluated the benefits of locality on both administrative (uVM destroy and uVM migration) and I/O tasks. Recall that

TABLE II
MC'S NEEDS FOR THREE CLOUD MANAGEMENT SYSTEMS.

	OpenStack	OpenNebula	Eucalyptus
# vCPUs	2	2	1
RAM (GB)	2	1.85	1.5

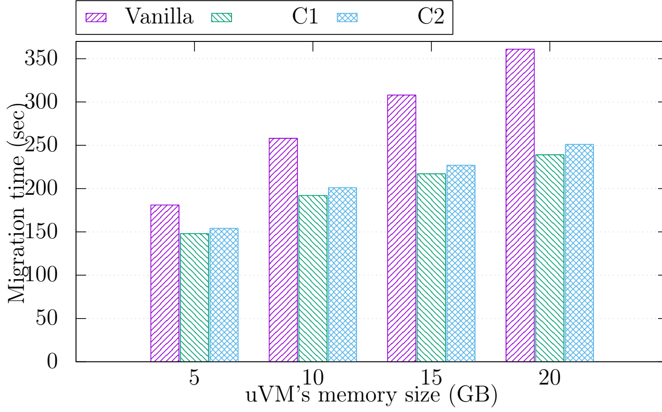

Fig. 8. Multi-threaded migration process (lower is better). The uVM runs a memory intensive application.

we provide locality by enforcing the execution of SC's vCPUs on the same socket as the concerned uVM. In order to only evaluate locality benefits, SC's vCPUs do not use uVM's CPUs (unless otherwise specified).

1) Administrative task improvement: The effectiveness of our uVM destruction solution is obvious and has been well demonstrated in [8] (Subsequently, we do not claim this innovation and we recommend the reader to refer to [8]). We focus on the novel multi-threaded migration solution we propose, whose effectiveness is not obvious due to the management of dirty pages. To this end, we evaluated our solution when the migrated uVM runs an intensive *Memory* benchmark. We experimented with different uVM memory sizes (4GB-20GB). For every experiment, the uVM's memory is equally spread over the two NUMA nodes of the server. We considered two situations (noted C_1 and C_2): in C_1, migration threads do not share the uVM's CPUs (assuming there are free CPUs available on the socket for migration) while they do in C_2 (assuming there aren't any available free CPU, the socket which runs the uVM being fully occupied). Fig. 8 presents the evaluation results (lower is better). We can see that our solution outperforms *Vanilla* in all experimented situations. The improvement is most important with large uVMs (up to 33% in C_1, see Fig. 8). Intuitively, the reader can imagine that the improvement will also increase when the number of sockets hosting the uVM's memory increases. C_1 slightly outperforms C_2 (up to 4%), justifying our strategy to only use the uVM's CPUs when there is no free CPU on the involved sockets.

2) I/O task improvement: We evaluated the benefits of running pVM's tasks that are involved in a uVM's I/O activity on the same socket as that uVM. We evaluated both network

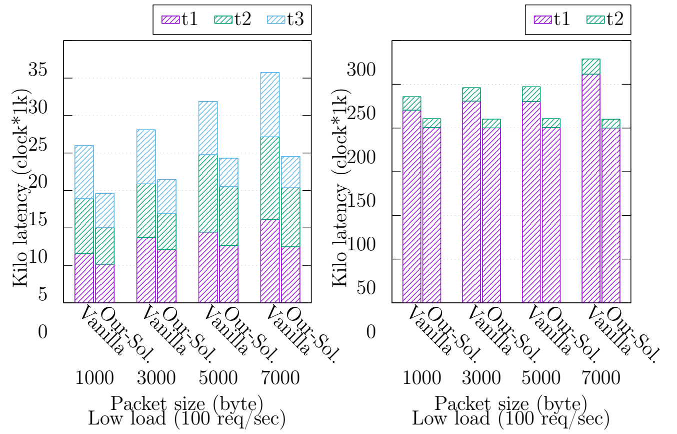

Fig. 9. Packet reception (left) and packet emission (right) improved by locality. (lower is better)

and disk activities. Concerning the former, we evaluated each version presented in Section IV-C, namely: *hyperLB* at hypervisor level; OAPI, NAPI-1 or NAPI-n at the pVM level. For these evaluations, we used a constant low load (100 req/sec).

Packet reception with a single uVM

The uVM is configured with 4vCPUs (all located on the second socket) and 4GB memory. Another server runs *Netperf*. We are interested in the following metrics: (t_1) the treatment duration before the backend invocation (this includes r_{3-6} steps, see Fig. 7); (t_2) the time taken by the backend to inform the frontend (step r_7); and (t_3) the time taken by the frontend to transfer the packet to the application's buffer. Fig. 9 left presents the evaluation results of each t_i while varying packet size. Here, all versions provide almost the same values, thus they are shown under the same label (*Our-sol.*) in Fig. 9 left. We can see that our solution minimizes all t_i in comparison with *Vanilla*, leading to a low $t_1 + t_2 + t_3$ (up to 36.50% improvement).

Packet reception with multiple uVMs

The previous experiments do not show any difference between our implementation versions. The differences appear only when the server runs several uVMs. To this end, we performed the following experiment. The testbed is the same as above in which a second uVM (noted d_2, the first one is noted d_1) runs on the first socket (the same which runs the MC). The two uVMs receive the same workload. We are interested in d_1's performance and we compare it with its performance when it runs alone (as in the previous experiments). Fig. 10 left presents $t_1+t_2+t_3$ normalized over *Vanilla*. We can see that OAPI and NAPI-n outperform NAPI-1 by about 21%. This is because NAPI-1 (which does a batch treatment) fails in choosing the correct target uVM for many request in r_4. This is not the case neither in OAPI nor in NAPI-n which are able to compute the correct destination for each packet. Fig. 10 right presents the amount of errors (wrong destination) observed for each implementation version. Thus, in further experiments, "our solution" refers to OAPI or NAPI-n.

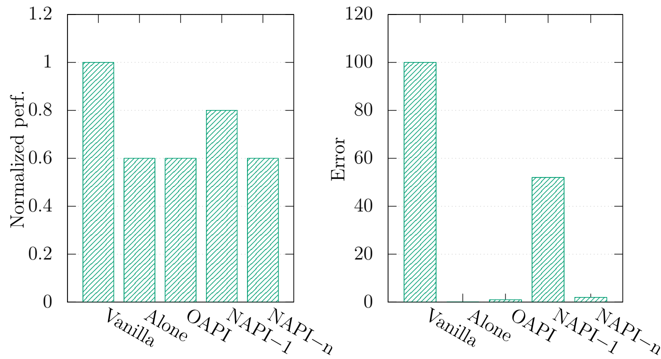

Fig. 10. Packet reception improved by locality: multiple uVMs evaluation. Results are normalized over *Vanilla* (lower is better).

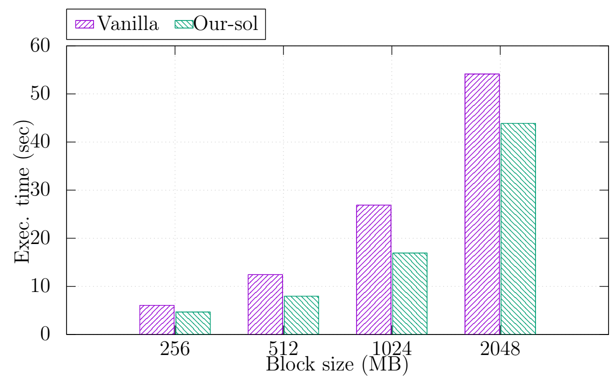

Fig. 11. Disk operations (from *dd*) improved by locality. (lower is better).

Packet emission
The testbed here is the same as in the experiment with packet reception and a single uVM, except the fact that *Netperf* runs inside the uVM. Remember that packet emission is agnostic about the different implementation versions of our solution (see Section IV-C). We are only interested in t_1 and t_2 (t_3 is constant). Fig. 9 right presents the results. We can see that thanks to locality, our solution improves (minimizes) t_1 by up to 42.33% for large packets in comparison with *Vanilla*. Compared with packet reception results, the improvement is more significant here because in the case of packet emission with *Vanilla*, memory is allocated (on emission) on the uVM socket and pVM's I/O tasks access the packet remotely. This is not the case with packet reception since the packet enters the machine via the pVM's socket (therefore, packet handling is more important in the pVM than in the uVM).

Disk operations
The testbed here is the same as above except the fact that the benchmark is *dd*. The uVM's hard disk is configured in write-through mode in order to avoid caching. The collected metric is the execution time. We evaluated different write block sizes. Fig. 11 presents the results. We can see that our solution outperforms *Vanilla*, especially with large blocks (up to 22% improvement).

Macro-benchmark improvement
We also evaluated the benefits of locality on macro-benchmarks. The testbed is the same as above in which the benchmark is replaced by a macro-benchmark. Fig. 12 left presents the results normalized over *Vanilla*. We can see that all the benchmarks are improved with our solution: about 25% for wordpress, 21% for Kernbench, and 30% for Magento.

D. Pay-per-use effectiveness

We validated the effectiveness of our architecture in charging to uVMs resources consumed by the pVM on their behalf. This includes demonstrating that MC's resource consumption remains constant regardless uVMs activities. We also evaluated the fairness of our solution, meaning that each uVM is charged proportionally to its activity. We only present the evaluation results for the packet reception experiment (which is the

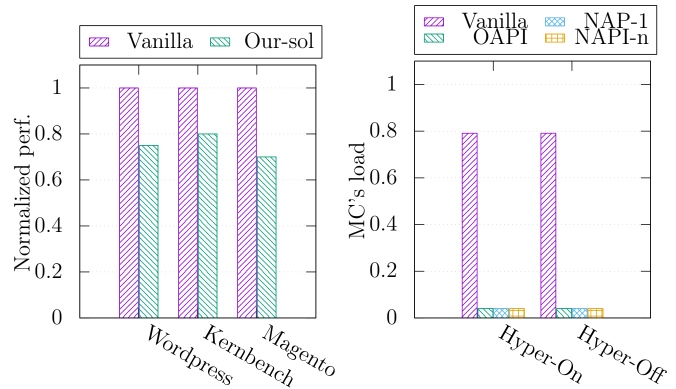

Fig. 12. (left) Macro-benchmarks improved by locality. The results are normalized over *Vanilla* (the first bar). (right) MC's CPU load.

most sensitive one, packet emission and disk operations being less tricky regarding pay-per-use). To this end, we used the same testbed as previously. The server under test runs three uVMs (noted vm_1, vm_2, and vm_3), each configured with two vCPUs. They share the same socket and each vCPU is pinned to a dedicated CPU. vm_1, vm_2, and vm_3 respectively receives 1000req/sec (which accounts for 10% of the total load), 3000req/sec (30% of the total load) and 6000req/sec (60% of the total load). In this experiment, we do not care about memory locality. We collected the following metrics: MC's CPU load, and the proportion of CPU time stolen by SC's vCPUs from each uVM (this represents the contribution of the uVM, used for fairness evaluation).

Fig. 12 right shows MC's CPU load consumption while Fig. 13 presents each uVM contribution. In Fig. 12, the ideal solution is the one which leads to no CPU consumption in the MC, meaning that the MC is not impacted by uVMs activities. The load reported for *Vanilla* is relative on the MC reserved capacity (e.g. 100% for *Vanilla* means that its CPU consumption is equivalent to the entire MC capacity). We can see that all our implementation versions are close to the ideal value, the best version being OAPI/NAPI-1 combined with *hyperLB-On* while the worst one is NAPI-n

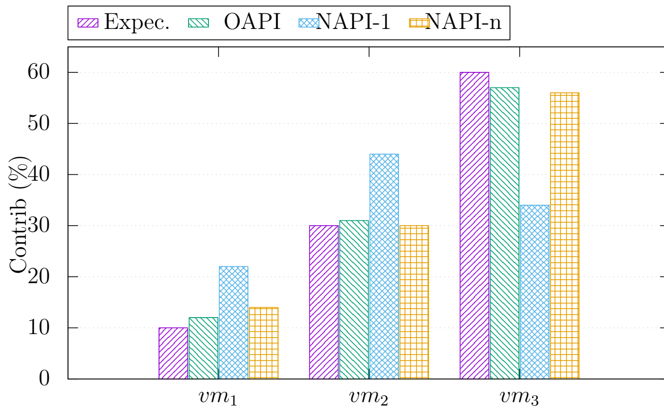

Fig. 13. Fairness of our solutions. (close to the first bar is better).

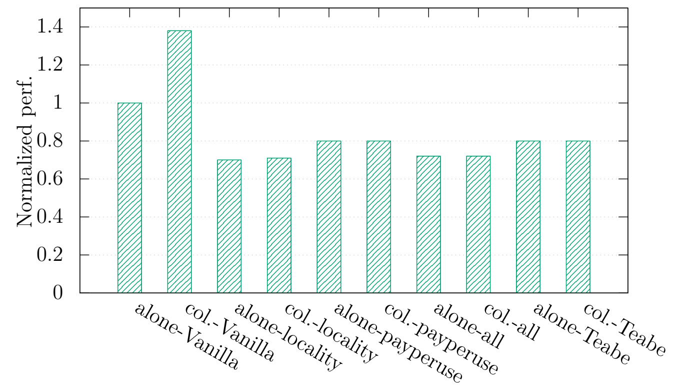

Fig. 14. Our solution with all mechanisms, Kernbench results. The latter are normalized over *Vanilla* when the benchmark runs alone. For predictability, compare <alone>-<solution> and <col.>-<solution>. For performance improvement, lower is better.

with *hyperLB-Off*. But the difference between these versions is very low, meaning that the activation of *hyperLB* is not necessary. In other words, r_3's CPU consumption is negligible. Therefore, in Fig. 13, we only present results with *hyperLB-Off* (*hyperLB-On* provides the same results). The ideal solution is the one which provides the expected result (the first bar). The latter corresponds to the proportion of the uVM's I/O traffic in the total traffic generated by all uVMs. We can see that except NAPI-1, both OAPI and NAPI-n ensure fairness.

E. All together

We evaluated our solution when locality and pay-per-use mechanisms are enabled at the same time. The testbed is the same as above where macro-benchmarks replace micro-benchmarks. We tested all possible colocation scenarios. Our solution is compared with *Vanilla* when the pVM is allocated the same amount of resources as the MC. Given a benchmark, the baseline value is the one obtained when it runs alone. Having already discussed the impact on performance, the interesting subject here is *performance predictability*. Several studies [7], [23] showed that this issue may occur when the pVM is saturated due to uVMs activities. We also compared our solution with [7] (called *Teabe* here). The latter presented a solution which ensures that the aggregated CPU time (including the one generated inside the pVM) used by a uVM cannot exceed the capacity booked by its owner. Fig. 14 presents the results for Kernbench using the best implementation version (NAPI-n with *HyperLB-Off*), all benchmarks running at the same time. In Fig. 14, each evaluation <situation>-<solution> is done in a given *situation* and with a given *solution*. Situation can be *alone* or *col* (for colocation of the 3 benchmarks) and Solution identifies the used solutions (*all* is our full solution). We can see that *Teabe* enforces predictability as our solution: in Fig. 14 *alone-all* is almost equal to *col.-all*; and *alone-Teabe* is almost equal to *col.-Teabe*. However, our solution improves application performance thanks to our locality enforcement contribution: *alone-all* (respectively *col.-all*) performs better than *alone-Teabe* (respectively *col.-Teabe*). As mentioned above, the improvement will increase when the number of sockets hosting the uVMs memory increases. Fig. 14 also shows the results reported in the previous sections so that the reader can easily appreciate on the same curve the impact of each contribution.

VI. Related work

Several research studies have investigated pVM management task improvements. Most of them have focused on uVM creation [8], [24]–[32] and migration [33]–[38] in order to provide reactive applications (quick elasticity) and consolidation systems. Very few of them exploit locality to improve these management tasks as we do.

Related to our scrubbing optimization, a Xen patch [39] was proposed to delay the scrubbing process and perform it during idle CPU cycles. Contrarily to our solution in which scrubbing is done synchronously with uVM destruction, [39] has the drawback of letting a completely arbitrary amount of time be spent before the memory is available again, introducing a lot of non-determinism. [8] presents a solution which is quite similar to the one we proposed.

Concerning live migration improvement, we didn't find any solution using a multi-threaded algorithm in order to execute in parallel and close to the migrated uVM's memory as we do.

Regarding I/O virtualization improvement, existing solutions either act at the hypervisor scheduler level [40]–[45] in order to minimize the scheduling quantum length thus minimizing I/O interrupt handling latency.

Very few researchers studied pVM resource utilization (on behalf of uVMs) from the pay-per-use perspective. To the best of our knowledge, [23] and [7] are the only studies which investigated pVM's resource partitioning as we do. [23] is limited to mono-processor machines while [7] leads to resource waste.

VII. Conclusion

This paper identified several issues related to the design of the pVM in today's virtualization systems. These issues arise from the fact that current pVMs rely on a standard OS (e.g., Linux) whose resource manager does not consider **the correlation between pVM's tasks and uVM's activities**. These issues lead to resource waste, low performance, performance unpredictability, and vulnerability to DoS attacks.

To take into account this correlation between virtual machines, we introduce *Closer*, a principle for implementing a suitable OS for the pVM. *Closer* promotes the proximity and the utilization of uVMs resources. It influences the design of the pVM with three main rules: **on-demand** resource allocation, **pay-per-use** resource charging and **locality** of allocated resources in a NUMA architecture. We designed a pVM architecture which follows *Closer* and demonstrated its effectiveness by revisiting Linux and Xen. An intensive evaluation of our implementation using both micro- and macro-benchmarks shows that this architecture improves both management tasks (destruction and migration) and application performance (I/O intensive ones), and enforces predictability.

VIII. Acknowledgment

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

References

[1] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265930

[2] 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*, ser. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[3] B. Bui, D. Mvondo, B. Teabe, K. Jiokeng, L. Wapet, A. Tchana, G. Thomas, D. Hagimont, G. Muller, and N. DePalma, "When extended para - virtualization (xpv) meets numa," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. Dresden, Germany: ACM, 2019, pp. 7:1–7:15. [Online]. Available: http://doi.acm.org/10.1145/3302424.3303960

[4] vmware.com, "Migration disabled with vmware vsphere sr-iov," Apr. 2018. [Online]. Available: https://goo.gl/d8MGD5

[5] Oracle.com, "Migration disabled with oracle vm server sr-iov," Nov. 2017. [Online]. Available: https://goo.gl/uneSHr

[6] Redhat.com, "Migration disabled with hyper-v sr-iov," Mar. 2015. [Online]. Available: https://goo.gl/qUtsQz

[7] B. Teabe, A. Tchana, and D. Hagimont, "Billing system CPU time on individual VM," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, 2016, pp. 493–496. [Online]. Available: https://doi.org/10.1109/CCGrid.2016.76

[8] V. Nitu, P. Olivier, A. Tchana, D. Chiba, A. Barbalace, D. Hagimont, and B. Ravindran, "Swift birth and quick death: Enabling fast parallel guest boot and destruction in the xen hypervisor," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. Xi'an, China: ACM, 2017, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050758

[9] Q. Huang and P. P. Lee, "An experimental study of cascading performance interference in a virtualized environment," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 43–52, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2479942.2479948

[10] Wordpress.org, "Wordpress." Aug. 2012. [Online]. Available: https://wordpress.org/plugins/benchmark/

[11] Pcisig.com, "Sr-iov." [Online]. Available: https://pcisig.com/specifications/iov/.

[12] Oracle.com, "Changing the dom0 memory size," Jun. 2014. [Online]. Available: https://docs.oracle.com/cd/E27300_01/E27308/html/vmiug-server-dom0-memory.html

[13] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 473–488. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872365

[14] Microsoft., "Microsoft's top 10 business practices for environmentally sustainable data centers 2010," Aug. 2009. [Online]. Available: http://tiny.cc/vro25y

[15] linux.die.net, "Irq balance," Jun. 2013. [Online]. Available: https://linux.die.net/man/1/irqbalance.

[16] U. Drepper, "What every programmer should know about memory," 2007.

[17] HewlettPackard, "Netperf," Jun. 2008. [Online]. Available: https://github.com/HewlettPackard/netperf

[18] kolivas.org, "Kernbench." Apr. 2014. [Online]. Available: http://ck.kolivas.org/apps/kernbench/kernbench-0.50/

[19] magento.com, "Magento," Mar. 2008. [Online]. Available: https://magento.com/

[20] openstack.org, "Openstack," Aug. 2017. [Online]. Available: https://www.openstack.org/

[21] openNebula, "Opennebula," Jul. 2017. [Online]. Available: https://opennebula.org/

[22] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2009.93

[23] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, ser. Middleware '06. Melbourne, Australia: Springer-Verlag New York, Inc., 2006, pp. 342–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=1515984.1516011

[24] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519067

[25] J. Zhu, Z. Jiang, and Z. Xiao, "Twinkle: A fast resource provisioning mechanism for internet services," in *2011 Proceedings IEEE INFOCOM*, April 2011, pp. 802–810.

[26] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. Newport Beach, California, USA: ACM, 2011, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952695

[27] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing vm checkpointing for restore performance in vmware esxi," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535463

[28] T. Knauth and C. Fetzer, "Dreamserver: Truly on-demand cloud services," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR 2014. Haifa, Israel: ACM, 2014, pp. 9:1–9:11. [Online]. Available: http://doi.acm.org/10.1145/2611354.2611362

[29] T. Knauth, P. Kiruvale, M. Hiltunen, and C. Fetzer, "Sloth: Sdn-enabled activity-based virtual machine deployment," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 205–206. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620765

[30] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh,

J. Ludlam, J. Crowcroft, and I. Leslie, “Jitsu: Just-in-time summoning of unikernels,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 559–573. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[31] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, “Rethinking the library os from the top down,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 291–304. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950399

[32] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, “My vm is lighter (and safer) than your container,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17, 2017.

[33] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, “Live virtual machine migration with adaptive, memory compression,” in *2009 IEEE International Conference on Cluster Computing and Workshops*, Aug 2009, pp. 1–10.

[34] S. Nathan, P. Kulkarni, and U. Bellur, “Resource availability based performance benchmarking of virtual machine migrations,” in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’13. Prague, Czech Republic: ACM, 2013, pp. 387–398. [Online]. Available: http://doi.acm.org/10.1145/2479871.2479932

[35] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, “Evaluation of delta compression techniques for efficient live migration of large virtual machines,” in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’11. Newport Beach, California, USA: ACM, 2011, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952698

[36] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe, “Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines,” in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’11. Newport Beach, California, USA: ACM, 2011, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952699

[37] Y. Abe, R. Geambasu, K. Joshi, and M. Satyanarayanan, “Urgent virtual machine eviction with enlightened post-copy,” in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 51–64. [Online]. Available: http://doi.acm.org/10.1145/2892242.2892252

[38] K. Kourai and H. Ooba, “Vmbeam: Zero-copy migration of virtual machines for virtual iaas clouds,” in *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016, Budapest, Hungary, September 26-29, 2016*, 2016, pp. 121–126. [Online]. Available: https://doi.org/10.1109/SRDS.2016.024

[39] Xen.org, “xen: free_domheap_pages: delay page scrub to idle loop.” May 2014. [Online]. Available: https://lists.xenproject.org/archives/html/xen-devel/2014-05/msg02436.htm

[40] 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*, ser. HPDC ’12. Delft, The Netherlands: ACM, 2012, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287080

[41] B. Teabe, A. Tchana, and D. Hagimont, “Application-specific quantum for multi-core platform scheduler,” in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys ’16. London, United Kingdom: ACM, 2016, pp. 3:1–3:14. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901340

[42] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, “vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core,” in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 243–254. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/xu

[43] D. Ongaro, A. L. Cox, and S. Rixner, “Scheduling i/o in virtual machine monitors,” in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’08. Seattle, WA, USA: ACM, 2008, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346258

[44] L. Zeng, Y. Wang, D. Feng, and K. B. Kent, “Xcollopts: A novel improvement of network virtualizations in xen for i/o-latency sensitive applications on multicores,” *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 163–175, June 2015.

[45] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, “Intel virtualization technology for directed i/o,” *Intel Technology Journal*, vol. 10, no. 03, pp. 179–192, August 2006.