IOMMU Para-Virtualization for Efficient and Secure DMA in Virtual Machines

Hongwei Tang[1,2,3], Qiang Li[2,3], Shengzhong Feng[1,3], Xiaofang Zhao[2,3] and Yan Jin[2,3]
[1] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
Shenzhen, 518055 - China
[e-mail: tanghongwei@ict.ac.cn, sz.feng@siat.ac.cn]
[2] Institute of Computing Technology, Chinese Academy of Sciences
Beijing, 100190 - China
[e-mail: liqiang@ncic.ac.cn, zhaoxf@ict.ac.cn, jinyan@ncic.ac.cn]
[3] University of Chinese Academy of Sciences
Beijing, 100049 - China
*Corresponding author: Hongwei Tang

Received August 14, 2016; revised November 3, 2016; accepted November 15, 2016; published December 31, 2016

Abstract

IOMMU is a hardware unit that is indispensable for DMA. Besides address translation and remapping, it also provides I/O virtual address space isolation among devices and memory access control on DMA transactions. However, currently commodity virtualization platforms lack of IOMMU virtualization, so that the virtual machines are vulnerable to DMA security threats. Previous works focus only on DMA security problem of directly assigned devices. Moreover, these solutions either introduce significant overhead or require modifications on the guest OS to optimize performance, and none can achieve high I/O efficiency and good compatibility with the guest OS simultaneously, which are both necessary for production environments. However, for simulated virtual devices the DMA security problem also exists, and previous works cannot solve this problem. The reason behind that is IOMMU circuits on the host do not work for this kind of devices as DMA operations of which are simulated by memory copy of CPU. Motivated by the above observations, we propose an IOMMU para-virtualization solution called PVIOMMU, which provides general functionalities especially DMA security guarantees for both directly assigned devices and simulated devices. The prototype of PVIOMMU is implemented in Qemu/KVM based on the virtio framework and can be dynamically loaded into guest kernel as a module, As a result, modifying and rebuilding guest kernel are not required. In addition, the device model of Qemu is revised to implement DMA access control by separating the device simulator from the address space of the guest virtual machine. Experimental evaluations on three kinds of network devices including Intel I210 (1Gbps), simulated E1000 (1Gbps) and IB ConnectX-3 (40Gbps) show that, PVIOMMU introduces little overhead on DMA transactions, and in general the network I/O performance is close to that in the native KVM implementation without IOMMU virtualization.

Keywords: IOMMU virtualization, para-virtualization, DMA security, virtio, simulated device

This work has been partly supported by the National Natural Science Foundation of China (NSFC) 61402444.

1. Introduction

Virtualization techniques help to improve the resource utilization and energy efficiency of large-scale data centers[26][27][28]. Based on virtualization, resources can be flexibly scheduled in finer granularity for different goals, such as QoS guarantees, energy consumption and etc[29][30]. Currently, virtualization is essential to resource management and scheduling in cloud data centers. However, in commodity virtualization platforms, there is a critical hardware component which has not been virtualized yet. It is I/O memory management unit (IOMMU). In general, IOMMU provides address remapping and address translation for DMA transactions. On this basis, it enables I/O virtual address space isolation between devices and memory access control on DMA transactions, by which it protects the system against malicious or buggy device drivers. In addition, it provides compatibilities with legacy devices which do not support memory address wider than 32 bits [1][10][11][20].

Without IOMMU virtualization, it adopts a simple method to support DMA of directly assigned devices in virtual machines [2][6]. At the VM launching phase, the VMM provisions host memory pages for all the guest memory and pins them until the termination of the VM. Moreover, it creates static DMA mappings in I/O page table, which map from guest memory pages to host memory pages, and are just the same as that in shadow page table. In the guest, all the devices reside in the memory address space, so that DMA engines of the devices access guest memory with guest memory address which is finally translated into host memory address by the IOMMU on the host. As all the guest memory is statically mapped in the I/O page table, malicious or buggy device drivers might tamper or corrupt guest memory using DMA, which is known as DMA security problem [7][8][21][22[23]. Furthermore, the ability of memory overcommitment of virtualization is missing in this case [4][10][12].

To solve this problem, [10] proposed vIOMMU, an IOMMU emulation solution which exposes an emulated IOMMU to the unmodified guests. The VMM intercepts and monitors access to the registers of the emulated IOMMU in guest, and manages the DMA mappings on the host IOMMU accordingly. The performance evaluations on vIOMMU shows that it introduces significant overhead and results in performance degradation on DMA. For example, the throughput of apache benchmark drops by about 50%. [9] presents an on-demand mapping strategy, in which the guest OS directly manages DMA mappings in the host I/O page table via hypercalls. It modifies guest devices drivers batching multiple map or unmap requests in a single hypercall to optimize performance.

Current solutions mainly have the following two deficiencies. On the one hand, the previous works focus only on directly assigned devices, while the DMA security problem also exists in simulated virtual devices. As for the latter, DMA transactions are simulated by memory copy between guest memory and host memory for virtual devices. So the hardware IOMMU on the host cannot provide access control for this kind of DMA transactions. On the other hand, I/O performance guarantee and guest OS compatiblity are both requisite for a practicle solution in production environments, however, the current solutions cannot meet this requirement.

In this paper, an IOMMU para-virtualization solution called PVIOMMU is proposed which provides efficient and secure DMA for both directly assigned devices and simulated virtual devices in the guest, and requires no code modification on the guest OS. In addition, it supports live migration and memory overcommitment for virtual machines. The main contributions of this paper are listed as follows.

1) We put forward the DMA security problem on simulated virtual devices, and gives a general IOMMU para-virtualization solution for both directly assigned devices and simulated virtual devices. For both kinds of devices, PVIOMMU provides isolation between I/O virtual

address spaces and memory access control on DMA transactions. A prototype of PVIOMMU is implemented based on the virtio framework [19] in KVM. The virtio frontend driver for PVIOMMU can be dynamically loaded into the guest kernel without modification or rebuilding of the guest kernel. Furthermore, to provide access control for DMA of virtual devices, a revised device model for Qemu is proposed which isolates the address space of the device emulator with that of the guest virtual machine.

2) We present the virtualization performance results of three kinds of network interface cards, including Intel I210 (1Gbps, directly assigned to the guest), simulated E1000 (1Gbps) and IB connectX-3 (40Gbps, virtual function directly assigned to the guest). **Table 1 a)** summarizes the performance differences between PVIOMMU and the native KVM implementation for Intel I210 and simulated E1000, while **Table 1 b)** shows that of IB. The results indicate that PVIOMMU achieves high efficiency close to that under native KVM without IOMMU virtualization, and can be used in production environments.

Table 1 a) Performance differences on ethernet NICs

	Intel I210	Simulated E1000
TCP_STREAM	0%	0%
TCP_RR	-15%	-7%
UDP_RR	-6%	-8%
Apache bench/10000reqs	-12%	+3%
Apache bench/100000reqs	-3.5%	+5%

Table 1 b) Performance differences on IB

	Send	Write
Latency(usec)	+0.013	+0.002
Bandwidth(MB/s)	-3	-5

The rest of the paper is organized in the following manner: Section 2 describes the DMA security problem in virtual machines. Section 3 gives a brief review of related works on IOMMU virtualization. Section 4 gives the design and implementation of PVIOMMU and optimization methods on both the guest side and the host side. Section 5 presents the experimental results, and finally Section 6 concludes the paper and mentions the future work.

2 DMA Security Problem in Virtual Machine

The framework adopted by KVM providing DMA support for devices in virtual machines is shown in **Fig. 1**. In general, there is no explicit IOMMU support in virtual machines. In guest OS, the configuration on IOMMU is usually set to “nommu”. In this case, for DMA buffer mapping requests from the upper layers (as in **Fig. 1** the DMA Mapping layer and the Device Driver layer), the “nommu” driver simply returns the guest physical memory addresses (abbr. GPAs) of the buffer as the device’s I/O virtual addresses. As a result, the GPAs are directly used by the device to perform DMA [4][12].

For directly assigned device (including virtual function of SRIOV-enabled device), the DMA engine is implemented as hardware circuits and the I/O virtual address (the same as GPA in this case) in DMA transaction is automatically translated to host physical memory address (abbr. HPA) by the IOMMU based on the I/O page table [1][6][20]. Because all the devices share the same I/O virtual address space which is just the guest memory address space, there is only one I/O page table on the host that is shared by guest devices. In essence, the I/O page table is nothing but a copy of the shadow page table, and is set up statically when creating the virtual machine.

In Qemu, the device simulator is implemented in the I/O thread which shares the same address space with guest VM. And DMA operations are implemented as memory copy between the guest memory and the virtual devices. As a result, the I/O thread is able to access any location of the guest memory, so does the simulative DMA engine. Before the simulative DMA engine performs memory copy, the memory virtualization module translates GPA in DMA transaction into host virtual memory address (HVA), which is used by the I/O thread to access the guest memory.

From the framework we find that, for both directly assigned device and simulated device, there are no I/O virtual address space isolation in the guest. Furthermore, neither validity checking nor access control for DMA transactions are provided on the host side, so that malicious or buggy guest device drivers could mange to tamper or corrupt the guest memory effortlessly. In summary, the current framework lacks of protections on DMA security. Especially for simulated device, to our knowledge, there are no related works on this problem currently.

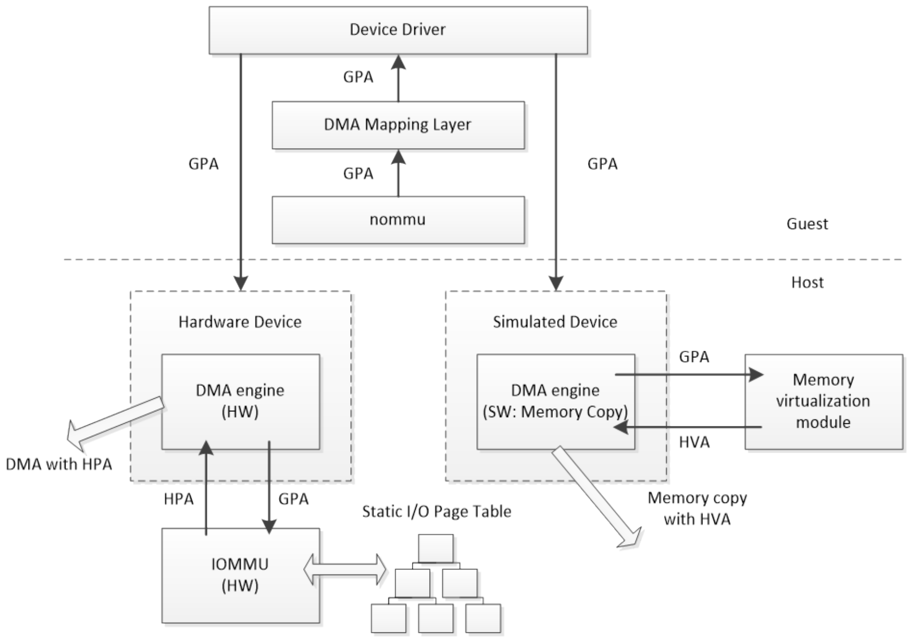


Fig. 1. Current framework of DMA support for VMs

3 Related Works

[10] presents a simulated Intel IOMMU by software, which has the advantages of complete transparency to the guest OS. However, it introduces considerable overhead on DMA transactions, especially a lot of context switches between the guest and the host. To improve performance, it utilizes extra host physical CPU cores to run the emulation codes of vIOMMU, and relaxes the security protections by delaying the retirement of IOTLB entries with different strategies. [9] proposes an on-demand DMA mapping strategy for directly assigned PCI devices where the guest OS proactively instructs the VMM to map or unmap DMA buffers in

a dynamic manner through hypercalls. It needs modifications on device drivers to adopt optimizations, such as batching map or unmap requests in a single hypercall to reduce overhead. As for performance results, it only evaluates the effect of the map cache inside the guest, in the perspective of both cache hit rate and CPU utilization, and no I/O performance results are published. [3] discusses the basic considerations on IOMMU support for virtualization. The basic design requirements that they emphasized include memory isolation, fault isolation and virtualized operating system support. They implement a proof-of-concept prototype on Xen while performance evaluations are less presented.

[14] discusses different IOMMU-based protection strategies in both the security perspective and the performance perspective. The single-use mapping strategy provides inter-guest protection with the greatest cost. While the shared mapping strategy tries to reuse established mappings rather than generating a new one for each I/O transaction. As the extreme of reuse, the direct mapping strategy allows maximum reuse of IOMMU mappings and can minimize runtime overhead. From the security perspective, the protection level drops as the performance improves for each strategy. The mapping strategy adopted by KVM can be categorized as the direct one [4][12], which requires pinning the guest's entire memory on the host, so that memory overcommitment is disabled. Furthermore, all the devices inside the guest reside in the same address space with the guest memory, and there is no intra-guest DMA isolation or protection provided. As in this case, the overhead on DMA mapping/unmaping is minimized, so we take it as the performance baseline for comparison with PVIOMMU.

In addtion, there are related works on performance evaluations and optimizaitons of IOMMU. [5] presents peformance characteristics of commodity IOMMUs in Linux, both on bare metal and Xen hypervisor. It evalutes the throughput and CPU utilization of disk I/O and network I/O workloads with and without IOMMU. [15] discusses the negative impact of IOTLB cache misses and propses software and hardware enhancements on IOMMU to reduce miss rate. [13] presents scalable IOMMU management design and address two bottlenecks, including assignment of I/O virtual addresses (abbr. IOVAs) and management of IOTLB.

4 System Design and Implementation

4.1 Overall architecture

The overall architecture of PVIOMMU is shown in **Fig. 2**. It is composed of the guest part and the host part. In the guest, a standard virtio frontend driver is presented as we called the PVIOMMU frontend, which provides an Intel-IOMMU like driver abstraction and general driver APIs (e.g. *dma_map_ops* in Linux) to the DMA Mapping Layer of the guest OS. As a result, there is no need to modify guest device driver to use PVIOMMU. Furthermore, the PVIOMMU frontend manages an isolated I/O virtual address space for each device, and allocates address ranges for DMA transactions dynamically.

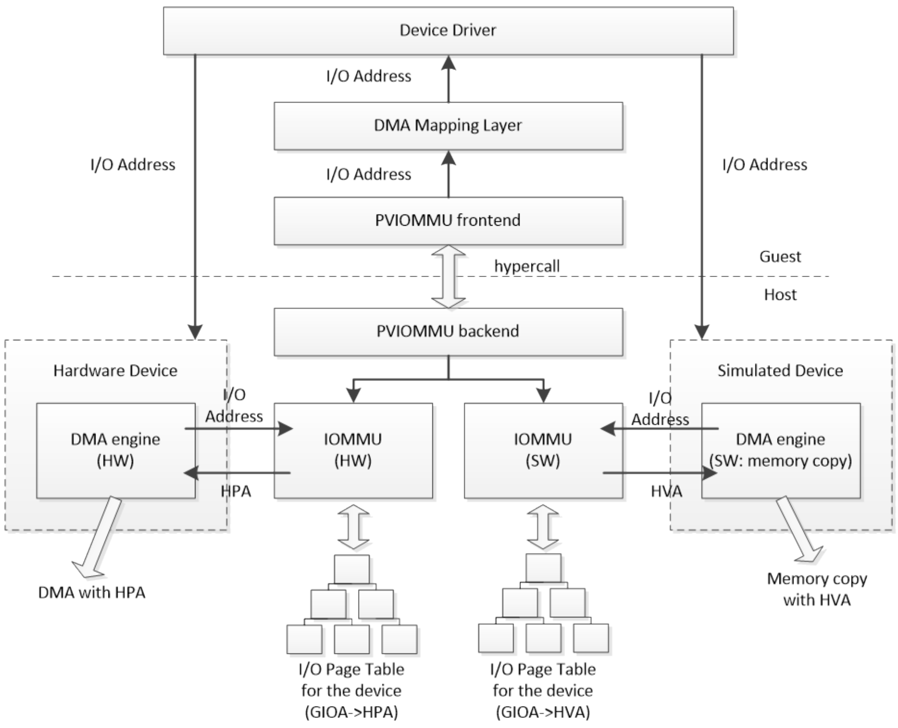


Fig. 2. Overal architecture of PVIOMMU

On the host side, it consists of I/O page tables, hardware IOMMU for directly assigned devices, software IOMMU for simulated devices, and the PVIOMMU backend driver. There is a dedicated I/O page table for each guest device. For directly assigned device, the I/O page table is referenced by hardware IOMMU providing mappings from devices' guest I/O virtual addresses (abbr. GIOVAs) to HPAs, while that for simulated devices is referenced by software IOMMU and provides mappings from GIOVAs to HVAs. The hardware IOMMU is commodity one such as Intel-IOMMU. While software IOMMU is implemented in the VMM and its design is detailed in subsection 4.2. The hardware and software IOMMUs provide I/O address translation and access control on DMA buffers based on I/O page tables. The PVIOMMU backend dynamically creates or tears down DMA mappings in I/O page tables. Furthermore, the backend manages the dynamic allocation of host page frames for guest DMA buffers, which enables thin-provisioning and overcommitment of guest memory.

The implementation of PVIOMMU is based on the virtio framework. Linux takes the *virtio_ring* as the transport implementation of virtio, which is composed of the descriptor array, the available ring and the used ring. The guest chains prepared buffers into the descriptor array, and indicates which descriptor chains are ready for use by the VMM in the available ring. While the VMM indicates which descriptor chains it has used in the used ring [19]. We adopt a single queue for both map and unmap requests. From the guest's view, a map request is composed of two parts: a read-only virtio_dma_mapping structure (as shown in **Table 2**) and a single write-only byte that indicates completion status. Similarly, an unmap request is also composed of two parts: a read-only virtio_dma_unmapping structure (as shown in **Table 3**) and a single write-only byte that indicates completion status. We put the two parts into two free entries of the descriptor table, and chain them together by the 'next' field, as is shown in **Fig. 3**.

Table 2. data structure of map request

```
struct virtio_dma_mapping {
        __u64  devid;      // the ID of guest device which this mapping is for
        __u32  type;       // 0 --- MAP
        __u32  len;        // len of the gpas/giovas array
        __u64  direction; // DMA_TO_DEVICE or DMA_FROM_DEVICE
        __u64  giovas[0];  // array of the GIOVAs to map
        __u32  gpas[0];    // array of the GPAs to map, giovas[i]<--> gpas[i]
};
```

Table 3. data structure of unmap request

```
struct virtio_dma_unmapping{
        __u64  devid;      // the ID of guest device which this unmapping is for
        __u32  type;       // 1 --- UNMAP
        __u32  len;        // len of the giovas array .
        __u64  giovas[0];  // array of the GIOVAs to unmap
};
```

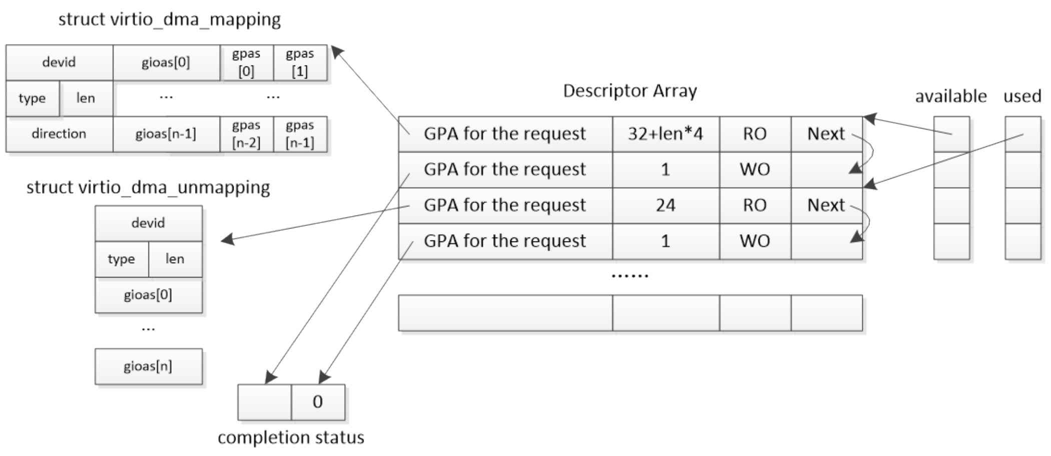


Fig. 3. Organization of PVIOMMU request queue

The PVIOMMU frontend calls *virtqueue_add* [12][19] function of Linux kernel to add request to the descriptor array, and kicks (*virtqueue_kick* [12][19] function of Linux kernel) the PVIOMMU backend. The *virtqueue_kick* function will merge multiple requests and notify the backend with only one hypercall, which reduces the overhead of context switching between the guest and the host. Then, the frontend calls *virtqueue_get_buf* [12][19] function iteratively to poll completed requests from the backend. When waiting for the completion, it utilizes the *cpu_relax* [12] function to do a lightweight polling which can also reduce power consumption. On the host side, a kernel thread named *vhost-pviommu* is created for each guest which services the guest's requests. Upon notification from the guest, the kernel thread unmarshals the request and checks the type field to identify which kind of request it is. '0' means map request while '1' means unmap request. After handling the request, the kernel thread writes completion status into the write-only byte, '0' means success while non-zero value means some fault and gives the fault number.

In summary, compared with the full-virtualization solution in [10], efficiency of interactions between the guest and the host is significantly improved because a lot of context switches are avoided. Moreover, in that case it has to traverse the guest I/O page table to get the GIOVAs and GPAs by recursively walking through the shadow page table [11], while in PVIOMMU, they are directly passed to the backend through the virtio queue.

4.2 DMA access control for simulated devices

In Qemu's device model, device simulation is implemented by the dedicated I/O thread which runs in the same address space of guest virtual machine from the host's view [25]. Therefore, the I/O thread can access any byte in the guest memory, and so does the simulative DMA engine. To enforce access control for DMA security, we revise the device model by separating the address space of the device simulator from that of guest virtual machine. In this model, the device simulator runs in a dedicated I/O process, and is dynamically granted access permissions on guest DMA buffers. The mechanism behind DMA access control for simulated devices is shown in **Fig. 4**. As also a simulated device, the software IOMMU is implemented in the I/O process, exposing API to translate GIOVAs to HVAs for the simulative DMA engine. The I/O page table for simulated device is placed in user space of the I/O process, where the software IOMMU and the PVIOMMU backend can both access with pointers directly.

- ***Create DMA mappings***

Given <GIOVA, GPA> pairs by the frontend, the backend firstly traverses the shadow page table (EPT on Intel systems) to translate the GPAs to HPAs. If any host page frame is not present, it will try to allocate one and fill the shadow page table. Then, the backend locates a free virtual memory area (VMA) in the virtual address space of the I/O process, and maps the host page frames corresponding to the HPAs into the VMA with proper access permissions, for example, read permission indicates DMA_TO_DEVICE and write permission indicates DMA_FROM_DEVICE. Finally, the backend creates mappings from GIOVAs to HVAs in the I/O page table and specifies the legitimate directions accordingly.

- ***Address translation***

Before accessing guest DMA buffers, the simulative DMA engine calls the software IOMMU to do address translation. The software IOMMU walks through the I/O page table to locate HVAs with GIOVAs as the indexes. Moreover, it also checks whether the DMA transaction is allowable against the DMA directions recorded in the I/O page table entries. Because address translation is on the critical path of DMA transaction, placing I/O page tables in user space can avoid the overhead of user-kernel context switching.

- ***Tear down DMA mappings***

First, the backend removes the mappings for specific GIOVAs in the I/O page table. Then it unmaps the corresponding VMA of the I/O process from the host page frames. In this way, the access permissions on the guest DMA buffer are revoked from the I/O process.

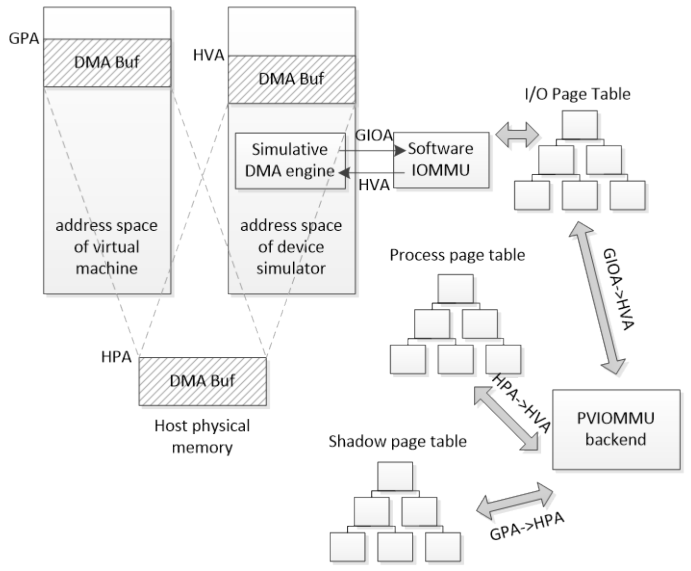


Fig. 4. Mechanism of DMA access control for simulated devices

4.3 Workflow of PVIOMMU

The workflow of creating DMA mappings with PVIOMMU is shown in **Fig. 5**:

Step 1: The device driver prepares memory buffer for this DMA transaction.

Step 2: The frontend allocates address ranges for the DMA buffer from the device's I/O address space.

Step 3: The mapping request is added to the available ring of the virtqueue. And if this is no pending requests, the virtio transportation will notify the backend to process by issuing a hypercall.

Step 4: The backend consumes request from the available ring.

Step 5: The backend locates the host page frames for the guest DMA buffer by walking through the shadow page table. If any page frame has not been allocated yet, it will try to allocate one.

Step 6: If the DMA is for simulated device, the backend will grant access permission on the guest DMA buffer to the device simulator by mapping the host page frames into the address space of the I/O process.

Step 7: The backend creates DMA mappings in the I/O page table for the device.

Step 8: The backend writes the completion status into the status byte, and adds the response to the used ring.

Step 9: The frontend gets the completion status from the used ring, and returns to the device driver.

Step 10: The DMA mapping process is finished, and the device driver proceeds with the DMA transaction.

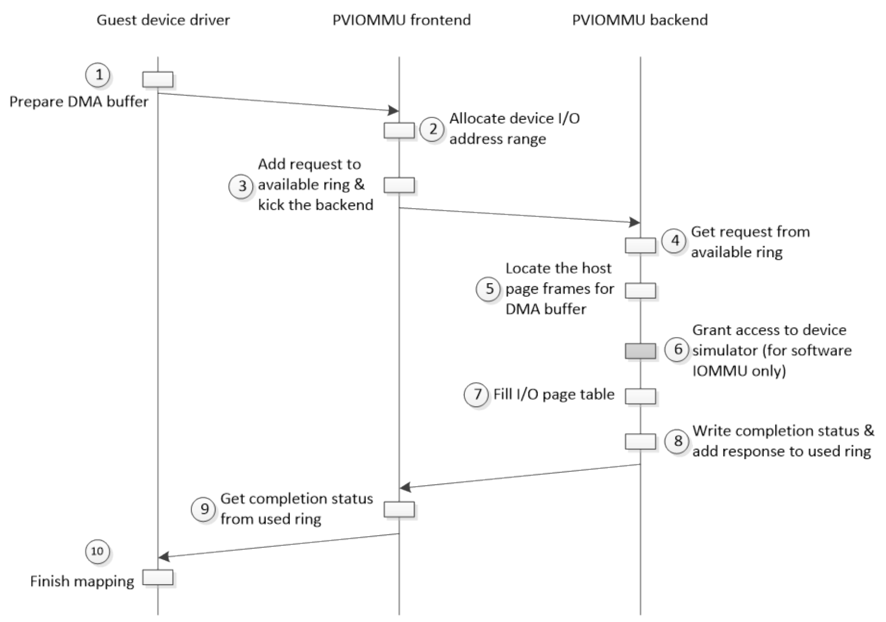

Guest device driver
PVIOMMU frontend
PVIOMMU backend
1 Prepare DMA buffer
2 Allocate device I/O address range
3 Add request to available ring & kick the backend
4 Get request from available ring
5 Locate the host page frames for DMA buffer
6 Grant access to device simulator (for software IOMMU only)
7 Fill I/O page table
8 Write completion status & add response to used ring
9 Get completion status from used ring
10 Finish mapping

Fig. 5. Workflow of PVIOMMU on creating DMA mappings.

Compared with the workflow of DMA mapping in guest serviced by native KVM, **Step 2** to **9** are added by PVIOMMU. From these steps, **Step 3** which might introduce guest-host context switches, and **Step 5** that walks through the shadow page table and may try to allocate page frames, are more costly operations. Motivated by this observation, we propose optimization methods to reduce the overhead in subsection 4.4 and 4.5.

4.4 Migration of PVIOMMU

Dynamic resource scheduling on virtualization platform relies mainly on two parts, the mechanism part which contains virtualization of hardware devices and live migration of virtual machine, and the policy part such as resource schedulers and scheduling algorithms. Essentially, PVIOMMU is a new virtual hardware device of virtual machine, so live migration of the virtual device is an essential function to support dynamic resource scheduling.

The states of PVIOMMU that should be migrated mainly include the general virtio states and established DMA mappings. The general virtio states such as virtqueues are migrated using the interfaces provides by Qemu/KVM [12]. Currently, directly assigned devices do not support live migration because internal information of such devices cannot be migrated. Accordingly, we focus on the migration of established DMA mappings for simulated devices. Because HVA in I/O page table is local address in I/O process' address space, it will be invalid on the destination host. To migrate DMA mappings, an intermediary mapping structure in guest view is generated from the host I/O page table on the source host. The intermediary mapping structure is shown in **Table 4**. Each such structure describes a VMA for guest DMA mappings in the I/O process. After the intermediary mapping structures are copied to the destination host, DMA mappings are reconstructed in the I/O process' address space and a new I/O page table is generated. After the guest resumes, the fake DMA engine continues operation with new mappings specified in I/O page table.

Table 4. defination of the intermediary mapping structure

```
struct dma_mapping_guestview {
        __u64  devid;      // the ID of guest device which this mapping is for
        __u32  len;        // len of the gpas array
        __u32  direction; // DMA_TO_DEVICE or DMA_FROM_DEVICE
        __u64  giova_start, giova_end;  // start and end of the guest I/O virtual address space
        __u64  gpas[0];   // array of the GPAs to map
};
```

4.5 Guest-side optimization

We have traced the DMA mapping creation operations in the driver of Intel I210 Ethernet NIC inside the virtual machine when performing basic network micro-benchmarking by Netperf [16]. **Table 5** shows the statistics collected during the test.

Table 5. Summary of DMA mapping creation traces

Test Case	Test Duration (seconds)	DMA Mapping Create OPS. #	Count of Involved Pages (4KB-size) #	Reuse Rate Of each page #
TCP_STREAM	10	82864	95	872
TCP_RR	10	59535	80	744
UDP_STREAM	10	14	12	1
UDP_RR	10	63367	11	5761

Considering the reverse process, each create operation is paired with a remove operation, so the count of DMA mapping manipulations doubles. As described in the workflow in subsection 4.3, these operations are expensive and may affect I/O performance. From the table we also find that, the page frames for DMA buffers are heavily reused. This observation motivates us to adopt guest-side optimization to reduce overhead.

The basic idea of the optimization is to delay the actual removal of DMA mappings in the host I/O page table and give chances to subsequent DMA transactions to reuse them. The frontend preserves a dedicated cache for each device to store reusable DMA mappings. A reusable mapping is the one that the device driver has released but the frontend has not removed from the host I/O page table. When creating a DMA mapping, the frontend first searches in the cache for reusable mappings with the GPAs of the DMA buffer. If a reusable mapping is found, the corresponding GIOVA will be reassigned to the DMA buffer and the process of creating mapping on the host for it is omitted. Since I/O page table on host is indexed by I/O virtual address [12], we name the cache as Reverse Translation Cache (abbr. RTC).

The structure for the RTC is shown in **Fig. 6**. For search efficiency and space cost, the RTC is organized as a combination of an adaptive radix-tree (abbr. ART) [31] and a FIFO queue. ART is a variation data structure to solve the problem of excessive worst-case space consumption of radix tree by adaptively designing compact and efficient data structures for nodes. To store DMA mappings in the ART, GPA is used as the key with the span of 8 (for byte alignment), and GIOVA is used as the value. In our case, 6 types of inner nodes and 6 types of leaf nodes are adopt, which are described as in Table 6. Leaf nodes are extensions based on corresponding inner nodes with the addition of a REUSING bit for each DMA mapping. When the REUSING bit is set (1), it means that the cached mapping is currently being reused. Moreover, GIOVA (64 bits long) is stored in the child pointer array of leaf node.

A DMA mapping should be invalidated after it has been cached in the RTC for a limited time to reduce security risks. For this purpose, firstly a timer is employed to split time into cycles of fixed time duration (*T*). Secondly, to effectively track expiration time for each mapping with low overhead, the coarser-grain cycle is used as the basic time unit. Finally, the cycle when a mapping is added into the RTC is recorded, and after fixed number (*C*) of cycles it will be invalidated. In this manner, mappings added at the same cycle are invalidated together, that further reduces the overhead on interacting with the PVIOMMU backend. As shown in **Fig. 6**, a FIFO queue is used for assisting to implement mapping invalidation periodically. Element of the FIFO queue is composed of a pointer to leaf node and a GPA, which represents a cached mapping. A special separator element is added to the FIFO queue at each timeout of the timer. By this means, mappings added at different cycles are separated in the FIFO queue. At any time, only the unexpired mappings which are added at the latest *C* cycles are stored in the FIFO queue. Suppose the oldest mappings in the FIFO queue are added at cycle numbered as *N*, they are also at the head of the FIFO queue. When the timer timeouts, a new cycle numbered as *N*+*C* starts, and mappings added in cycle *N* are removed from the FIFO queue and the ART. And then, the frontend issues a single request to the backend to delete the mappings in batch mode. It's worth mentioning that, mapping that is currently being reused (with the corresponding REUSING bit set in leaf node) should only be removed from the FIFO queue and be kept in the ART, because when it is released it will be added into the RTC again. The extra 1-bit REUSING flag can reduce the cost of insertion and deletion for reused mappings. In our implementation, *T* is set to 10 seconds and *C* is set to 3, so a mapping might be cached in the RTC for [30, 40) seconds.

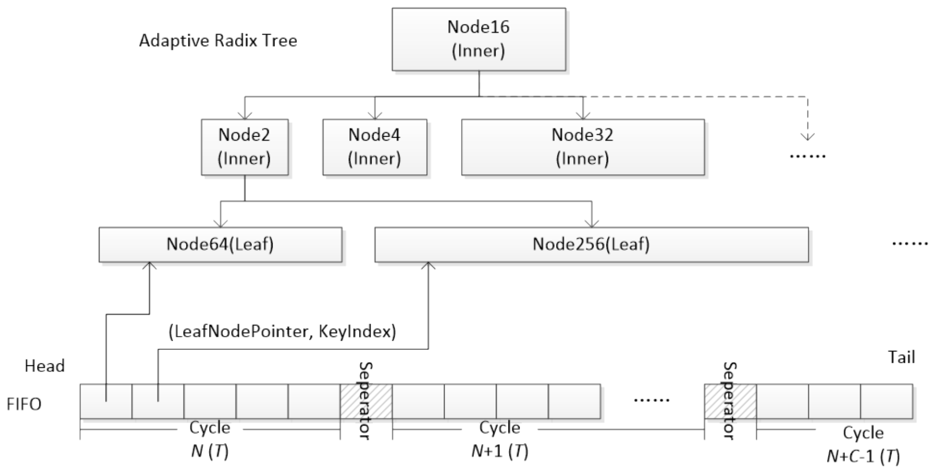


Fig. 6. Structure of the Reverse Translation Cache.

Table 6. Types of nodes (inner and leaf)

Node Type	Length of Key Array	Length of Pointer Array	# of Keys/ Pointers	Storage Organization	Search Method for Keys	Comparisons on key bytes for BinarySearch	Comparisons on key bytes for InsertSort
Node2	2	2	1~2	Sorted keys are stored in the key array, and corresponding	Binary search	<= 2	<=1

				child pointers are stored in the same position of the pointer array.			
Node4	4	4	3~4	The same as above	Binary search	<= 3	<=3
Node16	16	16	5~16	The same as above	Binary search	<= 5	<=15
Node32	32	32	17~32	The same as above	Binary search	<= 6	<=31
Node64	256	64	33~64	The key array is indexed with key bytes directly and stores indexes into the pointer array for corresponding key.	Indexing with key bytes	0	0
Node256	256	256	65~256	The pointer array is indexed with key bytes directly and no key array is provided.	Indexing with key bytes	0	0

- ***insert a DMA mapping into the RTC***

The pseudo-code of the insert algorithm is shown in **Fig. 7 a)** and **b)**. It starts from the root of the ART, recursively searches with key bytes of the given GPA, creates inner nodes and leaf node on the path if they have not yet existed, and finally stores the GIOVA in the corresponding position of the child pointer array in the leaf node. If the mapping is already in the ART, it is simply marked as being reused with the REUSING bit set. Binary search algorithm (`BinarySearch`) is used to search a specific key byte in nodes with less than or equal to 32 keys. While for nodes of type Node64 or Node256, the key byte is directly used as the index into the key array or the child pointer array respectively, so that no key comparisons are needed. If a key byte does not exist, it will be added into the corresponding node in sorted order (`InsertSort`). If a node is already full before the addition of the key byte, the node will be expanded to the next bigger node type (`Grow`). Finally, the pointer to the leaf node and the GPA are appended to the tail of the FIFO queue.

Algorithm1a): insert a DMA mapping into the ART

```
InsertART(GPA, GIOVA, root)
    return Insert(GPA, GIOVA, 63, root)
```

Fig. 7 a) pseudo-code for inserting a DMA mapping into the ART

Algorithm1b): insert a key byte into a node

```
Insert(GPA, GIOVA, bit, node)
if bit == 63
    key = GPA[bit, bit-3]
    bit = bit - 4
else
    key = GPA[bit, bit - 7]
    bit = bit - 8
if node.type is Node2, Node4, Node16 or Node32
    if (index = BinarySearch(key, node.keys, node.length)) < node.length
        if(bit <= 4)
            /*the key is aready exist in the leaf node indexed by index
```

```
             *Set the reusing bit of the mapping*/
            Set_bit(node.reusing, index)
            return ALREAD_EXIST
        else
            return Insert(GPA, GIOVA, bit, node.childpointers[index])
    else
       /*the key is not exist in the array, insert it*/
        if node.length is less than 2, 4, 16 or 32 respectively
            index = InsertSort(key, node.keys, node.length)
            if bit <= 4
                node.childpointers[index] = GIOVA
                Clear_bit(node.reusing, index)
                return SUCCESS
            else /*allocate a new smallest node*/
                node.childpointers[index] = new Node2()
                return Insert(GPA, GIOVA, bit, node.childpointers[index])
        else
        /*the node array is full, we should grow the node to a bigger one*/
            node = Grow(node)
            if node.type is Node4, Node16, Node32
                index = InsertSort(key, node.keys, node.length)
                if bit <= 4
                    node.childpointers[index] = GIOVA
                    Clear_bit(node.reusing, index)
                    return SUCCESS
                else
                    node.childpointers[index] = new Node2()
                 return Insert(GPA, GIOVA, bit, node.childpointers[index])
            else   /*node type is Node64 after growing*/
                index = node.keys[key]
                if bit <= 4
                    node.childpointers[index] = GIOVA
                    Clear_bit(node.reusing, index)
                    return SUCCESS
                else
                    node.childpointers[index] = new Node2()
                 return Insert(GPA, GIOVA, bit, node.childpointers[index])
else if node.type is Node64
    if  (index = node.keys[key]) < node.length
        if bit <= 4
            Set_bit(node.reusing, key)
            return ALREAD_EXIST
        else
            return Insert(GPA, GIOVA, bit, node.childpointers[index])
    else if node.length < 64
        node.keys[key] = index = node.length +1
        if bit <= 4
            node.childpointers[index] = GIOVA
            Clear_bit(node.reusing, index)
            return SUCCESS
        else
            node.childpointers[index] = new Node2()
            return Insert(GPA, GIOVA, bit, node.childpointers[index])
    else
        node = Grow(node)  /*node.type is Node256 after growing*/
        if bit <= 4
            node.childpointers[key] = GIOVA
            Clear_bit(node.reusing, key)
            return SUCCESS
        else
```

```
            node.childpointers[key] = new Node2()
            return Insert(GPA, GIOVA, bit, node.childpointers[key])
else /*node.type is Node256*/
    if node.childpointers[key] != NULL
        if bit <= 4
            Set_bit(node.reusing, key)
            return ALREAD_EXIST
        else
            return Insert(GPA, GIOVA, bit, node.childpointers[key])
    else
        if bit <= 4
            node.childpointers[key] = GIOVA
            Clear_bit(node.reusing, key)
            return SUCCESS
        else
            node.childpointers[key] = new Node2()
            return Insert(GPA, GIOVA, bit, node.childpointers[key])
```

Fig. 7 b) pseudo-code for recursively inserting a key byte into a node

- ***Search a DMA mapping in the RTC***

The pseudo-code of the search algorithm is shown in **Fig. 8 a)** and **b)**. It recursively searches from the root down to leaf node. The same as node insert, for nodes of type Node2, Node4, Node16 or Node32, binary search algorithm is adopted to search the given key byte in the node. While for nodes of type Node64, the key byte is used as the index into the key array, and the value specifies the index into the child pointer array. For nodes of type Node256, the key byte is the index into the child pointer array.

Algorithm2a): search in the ART for reusable DMA mapping with given GPA

```
SearchART(GPA, root)
    return Search(GPA, 63, root)
```

Fig. 8 a) pseudo-code for searching the ART

Algorithm2a): search in a node for the given key byte

```
Search(GPA, bit, node)
if bit < 12
    return node
if node == NULL
    return NULL
if bit == 63
    key = GPA[bit:bit-3]
    bit = bit -4
else
    key = GPA[bit:bit-7]
    bit = bit-8
if node is.type is Node2, Node4, Node16 or Node32
    if (index = BinarySearch(key, node.keys, node.length)) < node.length
        return Search(GPA, bit, node.childpointers[index])
    else
        return NULL
else if node.type is Node64
    if(index = node.keys[key]) <= node.length
        return Search(GPA, bit, node.childpointers[index])
    else
        return NULL
else
    return Search(GPA, bit, node.childpointers[key])
```

Fig. 8 b) pseudo-code for recursively searching key byte in node

- ***Delete a DMA mapping from the RTC***

The pseudo-code of the delete algorithm is shown in **Fig. 9**. It is the reverse process of insert. It deletes the values indexed by the key byte from the key array and the child pointer array of the node, and shrinks the node if the number of the left elements is less than the lower limit for the node type (`Shrink`). If the node is of type Node2 and the only element is deleted, the node will be removed and its parent node will be updated accordingly. In addition, the corresponding element in the FIFO queue is also removed.

Algorithm3: delete a DMA mapping from the ART

```
Delete(node, GPA, bit)
if bit >= 64
    return SUCCESS
if node == NULL
    return SUCCESS
if bit == 60
    key = GPA[bit+3:bit]
else
    key = GPA[bit+7:bit]
if node.type is Node2, Node4, Node16 or Node32
    index = BinarySearch(key, node.keys, node.length)
    if index >= node.length
        return  NOT_EXIST
    /*move the elements from index to the tail one position forward*/
    Move1(node.keys, index)
    Move1(node.childpointers, index)
    parent = node.parent
    if node.length is 0, 2, 4, 16 for the types respectively
        Shrink(node) /*Node2->NULL,Node4->Node2,Node16->Node4,… */
        if node == NULL
            /*if the node is deleted, we should delete
             *the corresponding key byte in parent node */
            return Delete(parent, GPA, bit+8)
else if node.type is Node64
    if node.keys[key] >= node.length
        return NOT_EXIST
    node.length = node.length – 1
    if node.length == 32
        Shrink(node) /*Node64->Node32*/
    else
        node.childpointers[node.keys[key]] = NULL
        node.keys[key] = node.length
else /*node.type if Node256*/
    if node.childpointers[key] == NULL
        return NOT_EXIST
    node.length = node.length – 1
    if node.length == 64
        Shrink(node) /*Node256->Node64*/
    else
        node.childpointers[key] = NULL
```

Fig. 9. pseudo-code for deleting a DMA mapping from the ART

The most time-consuming operation for search/insert/delete is comparison on key byte. The analysis on time complexity is presented in **Table 7**. As is depicted in the table, the time complexity has no relationship with the number of DMA mappings cached in the RTC. Instead, it is related to the height of the ART which depends on the width of GPA. The theoretical maximum length of GPA is 64 bits, and currently the addressing capability of mainstream

CPUs is no more than 48 bits. Moreover, DMA mapping is in page (4096 bytes) granularity, so the lowest 12 bits of GPA is of no use for identifying a mapping. For scalability, suppose the size of GPA is 64 bits, the valid key length is 52 bits and the height of the ART is 7. As is shown in Table 7, the best-case and worst-case time consumptions are all constant.

Table 7. Time complexity analysis

Operation	Best Case	Best-Case Time Complexity	Worst Case	Worst-case Time Complexity
SearchART	All the nodes on the searching path are with type of Node64 or Node256	No comparisons on key bytes	All the nodes on the searching path are with type of Node32	42 comparisons on key bytes.
InsertART	All the nodes on the searching path are with type of Node64 or Node256	No comparisons on key bytes	All the nodes on the searching path are with type of Node32 and all the nodes donot exist on the path.	42 (BinarySearch) + 217 (InsertSort) = 259 comparisons on key bytes.
Delete	All the nodes are with type of Node64 or Node256.	No comparisons on key bytes	The leaf node is of type Node32.	6 comparisons on the lowest key byte.

The space consumption of each type of node varies. For each inner or leaf node, there should be a 13-bytes extra space to store the type of the node (node.type, 1 byte), the number of valid keys/pointers in the node (node.length, 2 bytes), the pointer to the parent node (node.parent, 8 bytes). Especially for leaf nodes, extra space for REUSING bits is needed. For example, leaf nodes of type Node2 or Node4 need a single byte, while that of type Node16 needs 2 bytes, and so forth. The analysis on space complexity is presented in **Table 8**. The best space utilization can be achieved in that all the inner and leaf nodes are of type Node256 and there are no NULL pointers on each node. In this case, the average space consumption for each key byte is 8.1 bytes on inner nodes and 8.2 bytes on leaf nodes respectively. To optimize the space consumption, the PVIOMMU frontend tries to allocate adjacent guest pages for DMA buffers, so that more Node256 nodes are used.

Table 8. Space consumption of each types of nodes

Node Type	Total Size of Inner Node (Bytes)	Total Size of Leaf Node (Bytes)	Minimum Number of Keys	Maximum Number of Keys	Average Space Consumption with the Least Keys (Inner/Leaf)	Average Space Consumption with the Most Keys (Inner/Leaf)
Node2	2*1+2*8+13 =31	2*1+2*8+13+1=32	1	2	31/32	15.5/16
Node4	4*1+4*8+13=49	4*1+4*8+13+1=50	3	4	16.3/16.7	12.3/12.5
Node16	16*1+16*8+13=157	16*1+16*8+13+2=159	5	16	31.4/31.8	9.8/9.9
Node32	32*1+32*8+13=301	32*1+32*8+13+4=305	17	32	17.7/17.9	9.4/9.5
Node64	256*1+64*8+13=781	256*1+64*8+13+8=789	33	64	23.7/23.9	12.2/12.3
Node256	256*8+13=2061	256*8+13+32=2093	65	256	31.7/32.2	8.1/8.2

Finally, the lifecycle of DMA mapping with RTC in guest is described as follows.

Step 1: To create a DMA mapping in guest, the frontend first searches the RTC to find reusable mapping entries. For GPAs found in the RTC, the corresponding mappings are reused and marked as REUSING. While for GPAs that are not in the RTC, the frontend allocates new GIOVAs and creates new mappings in the host I/O page table by issuing a map request to the backend. Upon the backend completes the request, the corresponding mappings in the host I/O page table are available.

Step 2: The device driver performs DMA on the mappings.

Step 3: When a DMA mapping is released by the device driver, it is added into the RTC.

Step 4: When the timer timeouts, expired mappings are removed from the RTC. And then, the frontend issues a single request to the backend to remove the mappings from the host I/O page table in batch mode.

Step 5: If a mapping is reused before invalidated, it will be marked as REUSING, so that it will not be removed from the ART until it is released.

4.6 Host-side Optimization (HOPT)

On the critical path for creating a DMA mapping, the backend mainly completes two operations: walking through the shadow page table to translate GPAs into HPAs, and trying to allocate page frames if not present. Based on the above observation, we adopt the following two measures to further reduce the overhead.

- ***Pre-allocated page pool***

For each guest, the backend preserves a page pool to accelerate the process of dynamic page allocation. In our implementation, the size of the page pool is 1M bytes. As is shown in **Fig. 10**, a ring with 256 pointers to the pages in the pool is used for fast allocation of pages. In this case, pages are allocated in a sequential manner from the head of available pages pointers until to the tail.

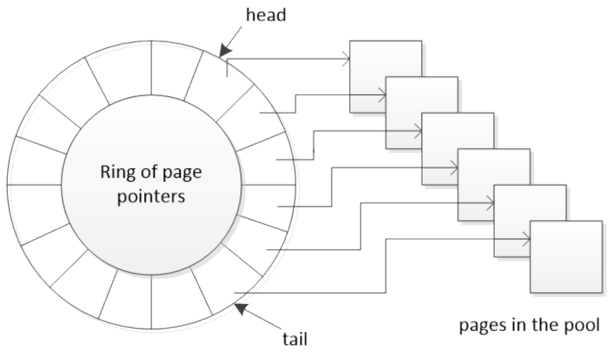


Fig. 10. Ring of pointers to pages in the pool

When host page frames for the given GPAs are not present, the backend firstly tries to fetch pages from the page pool. If there are not enough pages, it will resort to the host memory allocator. Moreover, the backend monitors the usage of the page pool. If the amount of available pages drops below the predefined threshold, in our implementation 64 pages, it requests new pages from the host memory allocator and appends the pointers at the tail of the ring. The space consumption of this optimization is 1026K bytes (1Mbytes + 256*8bytes), and the time of page allocation from the pool is negligible.

- ***Cache of the pointer to the last referenced page table***

To create DMA mappings, the backend translates GPAs to HPAs by walking through the shadow page table repeatedly. This takes hundreds to thousands of clock cycles to complete the operation. Based on the locality principle, we adopt a cache for the GPA prefix and the pointer to the last level (for example, the fourth level on 64-bit Linux) shadow page table that is referenced most recently, to reduce the overhead on shadow page table walking. To translate GPA to HPA, the backend first compares bits 20~63 of the GPA with the GPA prefix. If they match, it locates the entry with bits 12~19 of GPA as the index into the shadow page table referenced by the pointer. If the entry is already present, it just returns the corresponding HPA. Otherwise, it allocates a page from the page pool and fills the entry. If not match, it will walk through the shadow page table as usual, and finally updates the cache with the new GPA prefix and pointer. The space consumption of this optimization is 8 bytes for the pointer cache, and the time complexity is $O(1)$ because one comparison is required for translating each GPA.

5 Experimental Evaluations

5.1 Methodology

- ***Experimental Setup***

We use KVM hypervisor and CentOS 7.1.1503 running Linux 3.10.0 for both host and guest. Our experimental setup is comprised of two Sugon I620-G20 servers, which are dual-socket, 10-cores per socket server equipped with Intel Xeon E5-2650 V3 CPUs running at 2.3GHz. The server (host) includes 32GB of memory, an Intel I210 1Gb NIC and a Mellanox IB ConnectX-3 40Gb HCA card. The virtual machines are configured with 4 VCPUs and 8GB memory.

- ***Benchmarks***

For Ethernet, we use Netperf to evaluate network throughput and latency. In addition, we use Apache Bench [17] to evaluate the web service performance. For IB, we use the built-in tools and IMB 4.1 [18] to measure communication performance.

- ***Testbed system configurations***

We take the network I/O performance of virtual machines serviced by native KVM as the baseline. As introduced in Section 2 and Section 3, the native KVM implementation on IOMMU support for virtual machines adopts the direct mapping strategy with lowest overhead and optimal performance, but no DMA security guarantees. Our goal is to provide a performance as close as possible to the baseline on the premise that DMA security is guaranteed. So we evaluate four system configurations, including PVIOMMU with no optimization (abbreviated as PVIOMMU), PVIOMMU with reverse translation cache (abbreviated as PVIOMMU+RTC), PVIOMMU with RTC and host-side optimizations (abbreviated as PVIOMMU+RTC+HOPT), and the baseline in the native KVM (abbreviated as BASELINE).

5.2 Benchmark results

For Ethernet, performance is measured on Intel I210 NIC which is directly assigned to the guest, and simulated E1000 Ethernet NIC. While for IB, a virtual function of HCA card [24] is assigned to the guest.

5.2.1 Ethernet-Intel I210

The results of Netperf-TCP_STREAM test on Intel I210 are shown in **Fig. 11**. There is no significant difference among the four configurations at the aspect of throughput. Compared with the BASELINE, para-virtualization of IOMMU increases CPU utilization by about 9%. With optimizations of RTC and HOPT, CPU cycles consumed by PVIOMMU are reduced obviously, and the CPU utilization drops close to that of the BASELINE.

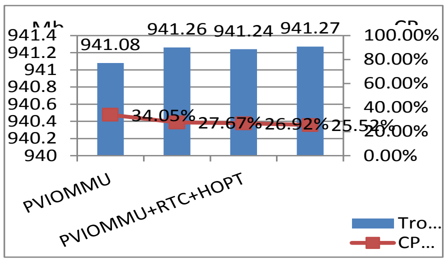


Fig. 11. TCP Throughput and CPU utilization of I210.

Fig. 12 a) presents the performance of exchanging requests and responses between the Netperf client and server, with packet size of 1 byte. The y-axis on the left side means amount of transactions per second, the more the better. While the y-axis on the right side means one-way communication latency, the shorter the better. As can be seen from the figure, compared with the BASELINE, the latencies increased by 21usec for TCP and 13usec for UDP when PVIOMMU is adopted, and the optimization techniques (RTC+HOPT) can further shorten the latency gaps to 13usec and 4usec for TCP and UDP respectively. Furthermore, RTC improves the latency dramatically, while HOPT has little promotion. **Fig. 12 b)** gives the very same data but normalized against the BASELINE. Moreover, comparisons on CPU utilization under the four configurations are presented as well. As can be seen from the curves (CPU_UTIL_TCP_RR and CPU_UTIL_UDP_RR), PVIOMMU incurs extra CPU utilization, and the optimization methods help to reduce CPU consumption.

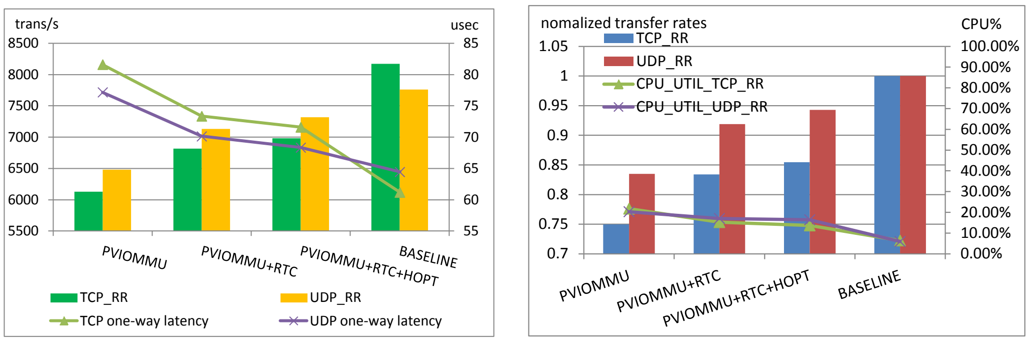


Fig. 12 a) TCP/UDP request-response rate and one-way latency. **b)** Transfer rates normalized to the BASELINE and CPU utilizations.

In the Apache Bench (ab) test, it simulates a simple web browsing scenario that the client requests a static page from the server. Fig. 13 a) and b) present the results of ab test with 100 concurrent connections. Fig. 13 a) shows different absolute request rates that can be achieved with 10,000 and 100,000 total requests. As can be seen from the figure, RTC is the most significant optimization for this metric. By normalized against the BASELINE, we can see from Fig. 8 b) that, by optimizing on PVIOMMU, the request rate can achieve 96% of the BASELINE.

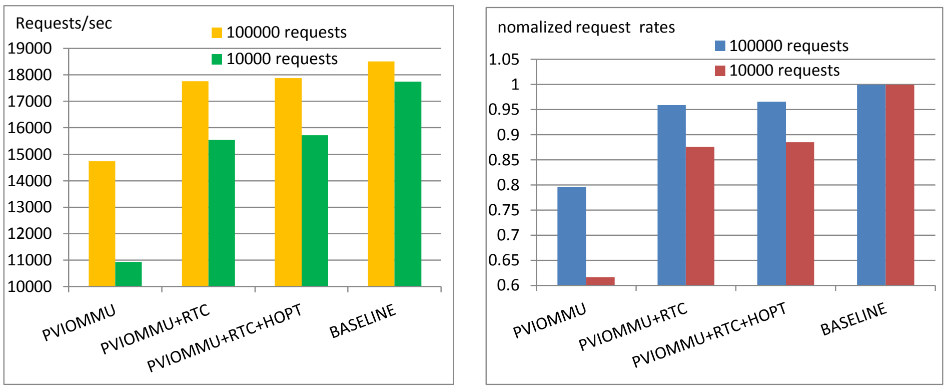


Fig. 13 a) Apache Bench test. **b)** The results normalized to the BASELINE.

5.2.2 Ethernet- Simulated E1000

Fig. 14 presents the TCP throughput and CPU utilization of simulated E1000 NIC. As seen from the figure, RTC optimization can boost the bandwidth to very close to the BASELINE. As the NIC is simulated by software, more CPU cycles are consumed when transferring packets compared with the directly assigned NIC. Although PVIOMMU increases CPU utilization, RTC and HOPT optimizations reduce that to an even lower level than the BASELINE. That is because that, under the BASELINE case the driver for E1000 in the guest utilizes bounce buffers to move data between DMA buffers and user-provided buffers which causes CPU utilization increasing.

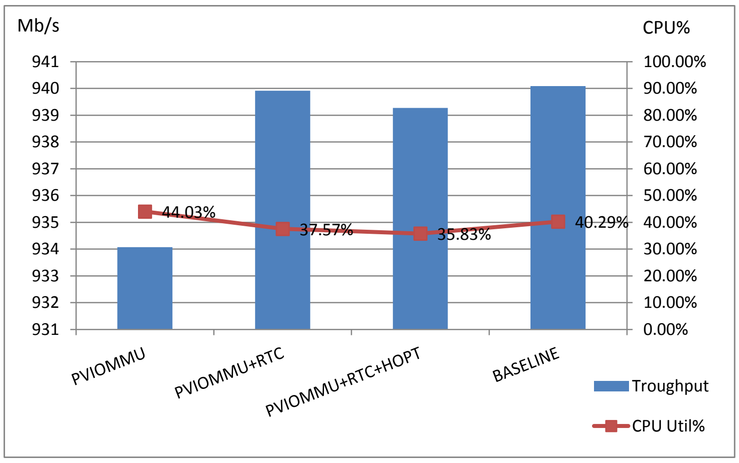
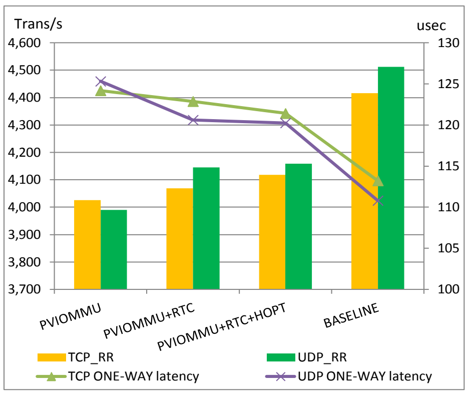


Fig. 14 TCP Throughput and CPU utilization of simulated E1000

Fig. 15 a) and b) show the results of TCP_RR and UDP_RR tests on simulated E1000. By adopting PVIOMMU without optimization, the TCP request-response performance can achieve about 91% of the BASELINE, while it only achieves about 75% on Intel I210 (shown in Fig. 12 b)). The reason is that hardware DMA engine is more efficient than simulated DMA engine, so the overhead of IOMMU virtualization is more significant in the former case. Furthermore, as seen from the curves (CPU_UTIL_TCP_RR and CPU_UTIL_UDP_RR) in Fig. 15 b), the CPU consumption of PVIOMMU is reduced to no more than 4% by adopting RTC and HOPT optimizations. Overall, the CPU consumption of PVIOMMU is moderate.



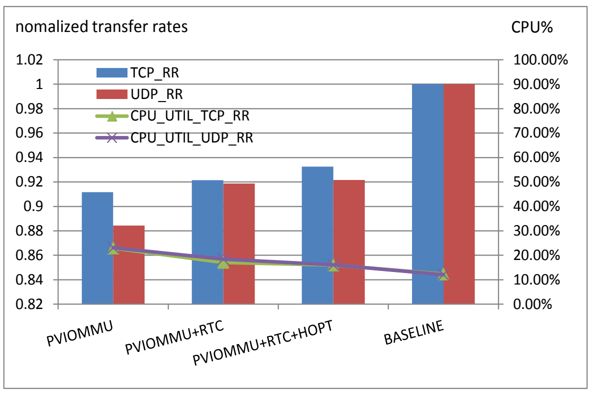
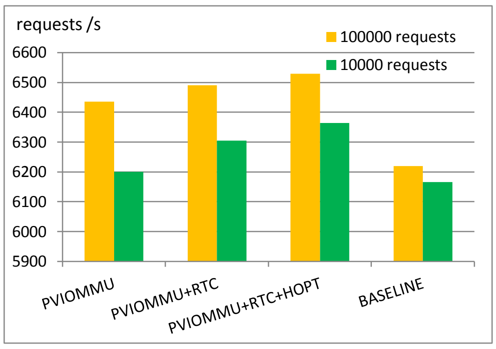


Fig. 15 a) TCP/UDP request-response and one-way latency. b) Transfer rates normalized to the BASELINE.

Fig. 16 a) and b) shows the ab test results on simulated E1000. The results reveal that it can get better performance with PVIOMMU compared to the native KVM implementation. In the latter case, the guest device driver uses software IOTLB to setup static DMA mappings and employs bounce buffer for legacy devices. The E1000 device driver uses a lot of bounce buffers to complete DMA transaction that introduces a large amount of extra memory copies and significant overhead. As a result, we can claim that PVIOMMU can improve performance for devices using bounce buffer.



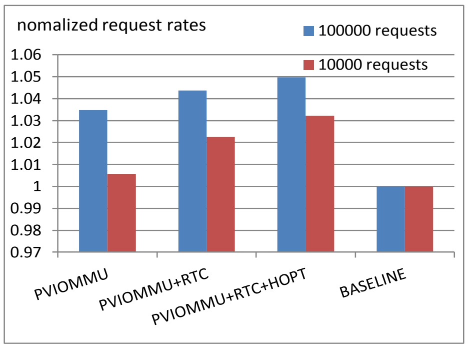
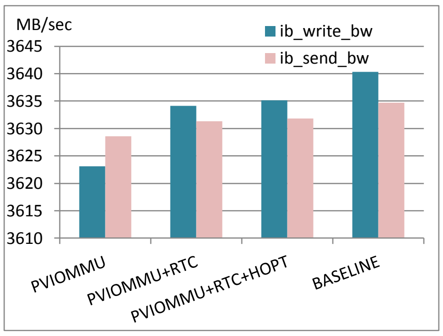


Fig. 16 a) ab request rates on simulated E1000. b) ab request rates normalized against the BASELINE.

5.2.3 IB

We use the IB verbs-level test programs (ib_write_bw, ib_write_lat, ib_send_bw and ib_send_lat) provided with OFED to benchmark the performance of IB. ib_write_bw and ib_write_lat measure the bandwidth and latency respectively using RDMA write transactions, while ib_send_bw and ib_send_lat using send transactions [32]. As **Fig. 17 a)** shows, among the four configurations, the disparity on bandwidth measured by ib_write_bw (17MB/s) is more obvious than that by ib_send_bw (6MB/s). The reason is that RDMA transaction needs more dynamic DMA mappings to access user-provided buffers, while send transaction uses pre-registered buffers with fixed DMA mappings. After optimization, the gap is narrowed down to about 5Mb/s. While the differences on latencies are very small among the four configurations as is shown in **Fig. 17 b)**.



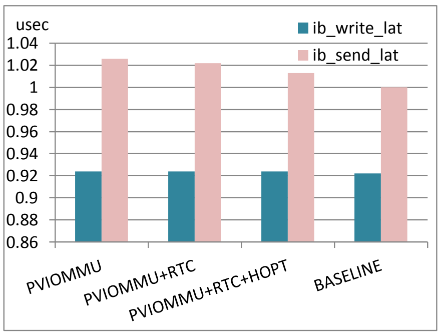
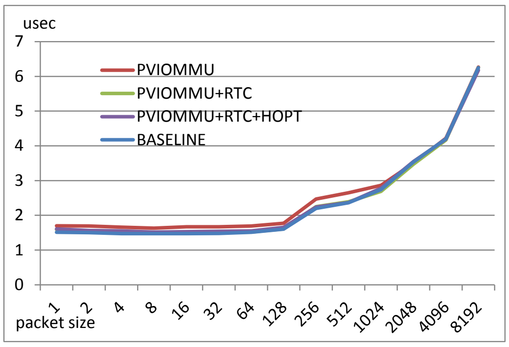


Fig. 17 a) IB bandwidth. **b)** IB latency.

The IMB PingPong test results are shown in **Fig. 18 a)** and **b)**. The four curves almost overlap, which indicates that the overhead of IOMMU virtualization is negligible on MPI communications.



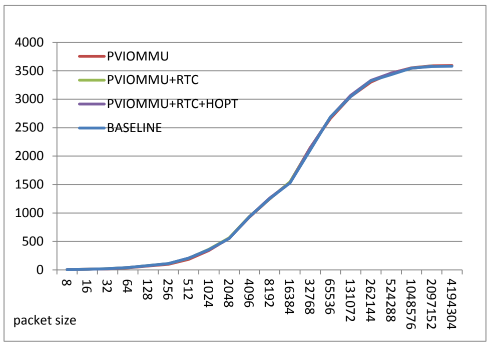


Fig. 18 a) latency results of IMB PingPong test. **b)** bandwidth results of IMB PingPong test.

6 Conclusion and Future Work

In this paper, we present PVIOMMU, an IOMMU para-virtualization solution to support high efficiency and secure DMA in virtual machines. A prototype is implemented in Qemu/KVM based on the virtio framework. Specially, to provide access control on DMA transactions of simulated devices, we revised Qemu's device model to separate the device emulator from the address space of the guest virtual machine. In addition, we adopt optimizations on PVIOMMU to further reduce virtualization overhead. On the guest side, the reverse translation cache for DMA mappings is employed to allow reusing. While on the host side, the pre-allocated page pool is adopted to accelerate page frame allocation for guest DMA buffers. Moreover, time for walking through the shadow page table is shortened by caching the pointer to the last referenced last-level shadow page table. Network performance evaluations on Intel I210 NIC, simulated E1000 NIC and IB ConnectX-3 card show that, PVIOMMU introduces little overhead on DMA transactions compared to the native KVM implementation, which lacks of protections on DMA security.

GPU is another kind of device that heavily relies on DMA operations. Our future work will extend the performance evaluation to GPU virtualization, for both simulated and hardware-assisted vGPU. And then, we will take performance tunings and optimizations based on the benchmarking results. In addition, we will dive deeper into resource scheduling on virtual machine with PVIOMMU, evaluate the influence of IOMMU para-virtualization on scheduling decisions, and study algorithms that can avoid instabilities on virtual machine migration.

Acknowledgments

We would like to thank the anonymous reviewers and editors for their valuable suggestions and comments to improve the quality of this paper.

References

[1] Intel, “Intel Virtualization Technology for Directed I/O Architecture Specification.” Article (CrossRef Link).
[2] B. Liu, L. Yang and X. Qin, “Research on Hardware I/O Passthrough in Computer Virtualization,” in *Proc. of the International Symposium on Computer Science*, 2010. Article (CrossRef Link).
[3] M. Benyehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick and E. Wahlig, “Utilizing IOMMUs for virtualization in Linux and Xen,” in *Proc. of OLS ’06: The 2006 Ottawa Linux Symposium*, pp. 71-86, 2006. Article (CrossRef Link).
[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin and A. Liguori, “KVM: the linux virtual machine monitor,” in *Proc. of Ottawa Linux Symposium*, pp. 225-230, 2007. Article (CrossRef Link).
[5] M. Benyehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer and L. van Doorn, “The price of safety: Evaluating IOMMU performance,” in *Proc. of OLS ’07: The 2007 Ottawa Linux Symposium*, pp. 9-20, 2007. Article (CrossRef Link).
[6] B. A. Yassour, M. Benyehuda and O. Wasserman, “Direct device assignment for untrusted fully-virtualized virtual machines,” vol. 54, pp. 150-156, Yehuda, 2008. Article (CrossRef Link).
[7] M. Becher, M. Dornseif and C. N. Klein, “FireWire: all your memory are belong to us,” in *Proc. of CanSecWest Applied Security Conference*, 2005. Article (CrossRef Link).
[8] R. Wojtczuk, “Subverting the Xen hypervisor,” in *Proc. of Black Hat*, 2008. Article (CrossRef Link).
[9] B. A. Yassour, M. Benyehuda and O. Wasserman, “On the DMA mapping problem in direct device assignment,” in *Proc. of SYSTOR 2010: the Haifa Experimental Systems Conference*. pp. 1-12, Israel, 2010. Article (CrossRef Link).
[10] N. Amit, M. Benyehuda, D. Tsafrir and A. Schuster, “vIOMMU: efficient IOMMU emulation,” in *Proc. of the 2011 USENIX conference on USENIX annual technical conference*, Portland, 2011. Article (CrossRef Link).

[11] M. Malka, N. Amit, M. Benyehuda and D. Tsafrir, “rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers,” in *Proc. of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Turkey, 2015. Article (CrossRef Link).
[12] Linux Kernel. https://www.kernel.org/.
[13] O. Peleg, A. Morrison, B. Serebrin and D. Tsafrir, “Utilizing the IOMMU scalably,” in *Proc. of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pp. 549-562, CA, 2015. Article (CrossRef Link).
[14] P. Willmann, S. Rixner, A. L. Cox, “Protection strategies for direct access to virtualized I/O devices,” in *Proc. of USENIX Ann. Technical Conf. (ATC)*, pp. 15-28, 2008. Article (CrossRef Link).
[15] N. Amit, M. Benyehuda and B. A. Yassour, “IOMMU: Strategies for mitigating the IOTLB bottleneck,” in *Proc. of Workshop on Interaction between Opearting Syst. & Comput. Archit. (WIOSCA)*, 2010. Article (CrossRef Link).
[16] Netperf. http://www.netperf.org
[17] Apache Bench. https://httpd.apache.org/docs/2.4/programs/ab.html
[18] IMB. https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[19] R. Russell, “virtio: towards a de-facto standard for virtual I/O devices,” *ACM SIGOPS Operating Syst. Review (OSR)*, vol. 42, pp. 95-103, 2008. Article (CrossRef Link).
[20] AMD, “AMD I/O Virtualization Technology (IOMMU) Specification (Revision 2.62),” February 2015. Article (CrossRef Link).
[21] P. Stewin and I. Bystrov, “Understanding DMA Malware,” in *Proc. of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, July 2012. Article (CrossRef Link).
[22] L. Duflot and Y. A. Perez, “Can You Still Trust Your Network Card?” in *Proc. of the 13th CanSecWest Conference (CanSecWest’10)*, 2010. Article (CrossRef Link).
[23] F. Sang, V. Nicomette, and Y. Deswarte, “I/O Attacks in Intel PC-based Architectures and Countermeasures,” in *Proc. of SysSec Workshop (SysSec’11)*, 2011. Article (CrossRef Link).
[24] J. Jose, M. Li, X. Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda, “SR-IOV support for virtualization on infiniband clusters: Early experience,” in *Proc. of Cluster Computing and the Grid, IEEE International Symposium on. IEEE Computer Society*, pp. 385-392, 2013. Article (CrossRef Link).
[25] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proc. of the USENIX Annual Technical Conference*, pp. 41-46, 2005. Article (CrossRef Link).
[26] C. Canali, and R. Lancellotti, “A class-based virtual machine placement technique for a greener cloud,” in *Proc. of 4th. Int. Conference on Green IT Solutions (ICGREEN 2015)*, 2015. Article (CrossRef Link).
[27] C. Canali, and R. Lancellotti, “Automated clustering of VMs for scalable cloud monitoring and management,” in *Proc. of the 20th International Conference on Software, Telecommunications and Computer Networks*, pp.1-5, 2012. Article (CrossRef Link).
[28] M. Shojafar, N. Cordeschi, D. Amendola, and E. Baccarelli, “Energy-saving adaptive computing and traffic engineering for real-time-service data centers,” in *Proc. of the IEEE International Conference on Communication*, pp. 1800-1806, 2015. Article (CrossRef Link).
[29] M. Shojafar, C. Canali, R. Lancellotti, and E. Baccarelli, “Minimizing computing-plus-communication energy consumptions in virtualized networked data centers,” in *Proc. of 21th IEEE/ACM ISCC*, pp. 1184-1191, 2016. Article (CrossRef Link).
[30] Z. Pooranian, M. Shojafar, R. Tavoli, M. Singhal, and A. Abraham, “A hybrid metaheuristic algorithm for job scheduling on computational grids,” *Informatica*, vol. 37(2), pp. 157-164, 2013. Article (CrossRef Link).
[31] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: ARTful indexing for main-memory databases,” in *Proc. of ICDE*, pp. 38-49, 2013. Article (CrossRef Link).
[32] OFED. https://www.openfabrics.org/downloads/OFED/

Hongwei Tang is a Ph.D. candidate at the University of Chinese Academy of Sciences. He received his B.S. degree from Nankai University (China) in 2006 and M.S. degree from University of Chinese Academy of Sciences in 2009. Currently, he is a senior member of China Computer Federation (CCF) and an associate professor at Institute of Computing Technology of Chinese Academy of Sciences (ICT, CAS). His research interests include cloud computing, virtual machine and operating system.

Qiang Li received the B.S. degree in computer science and technology in Shandong University in 2006, and Ph.D. degree in Institute of Computing Technology in 2012. He is an associate professor in Institute Computing Technology, Chinese Academy of Sciences. His research interests include HPC architecture and high performance communication.

Shengzhong Feng received his Ph.D. degrees in computer science from Beijing Institute of Technology in 1997. He was employed by Institute of Computing Technology, Chinese Academy of Sciences. From 2005 to 2007, he investigated gene chip algorithm design as a visiting professor in University of Toronto, Canada. Currently, he is the executive director of the China Mathematics Software Association and a committee member of High Performance Computing Association. He has published over 30 research papers, most of them are indexed by SCI/EI. His research interests include high performance computing, grid computing and bioinformatics.

Xiaofang Zhao received her Ph.D. degrees in computer science from Institute of Computing Technology, Chinese Academy of Sciences. Currently, she is the director of computer application research center in Institute of Computing Technology, Chinese Academy of Sciences. She is a senior member of China Computer Federation (CCF) and a member of Engineering and Technology Committee. Her research interests include computer architecture of next generation, security of computer system and information security.

Yan Jin received the B.S., M.S., and Ph.D. degrees in computer science from Harbin Institute of Technology, Harbin, China, in 2001, 2003, and 2008, respectively. He was a research fellow in department of Electrical and Computer Engineering, University of Nevada, Las Vegas (UNLV) from 2008 to 2011. Since 2012, he has been an associate professor in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include network security, cloud computing and cloud security.