J Supercomput (2012) 59:443–468
DOI 10.1007/s11227-010-0444-9

A low-overhead networking mechanism for virtualized high-performance computing systems

Jae-Wan Jang · Euiseong Seo · Heeseung Jo · Jin-Soo Kim

Published online: 26 May 2010

Abstract The use of virtualized parallel and distributed computing systems is rapidly becoming the mainstream due to the significant benefit of high energy-efficiency and low management cost. Processing network operations in a virtual machine, however, incurs a lot of overhead from the arbitration of network devices between virtual machines, inherently by the nature of the virtualized architecture. Since data transfer between server nodes frequently occurs in parallel and distributed computing systems, the high overhead of networking may induce significant performance loss in the overall system. This paper introduces the design and implementation of a novel networking mechanism with low overhead for virtualized server nodes. By sacrificing isolation between virtual machines, which is insignificant in distributed or parallel computing systems, our approach significantly reduces the processing overhead in networking operations by up to 29% of processor load, along with up to 36% of processor cache miss. Furthermore, it improves network bandwidth by up to 8%, especially when transmitting large packets. As a result, our prototype enhances the performance of real-world workloads by up to 12% in our evaluation.

J.-W. Jang · H. Jo
CS Dept., KAIST, Yuseong-gu, Daejeon, Republic of Korea

J.-W. Jang
e-mail: jwjang@calab.kaist.ac.kr

H. Jo
e-mail: heesn@calab.kaist.ac.kr

E. Seo
School of ECE, UNIST, Ulju-gun, Ulsan, Republic of Korea
e-mail: euiseong@unist.ac.kr

J.-S. Kim (✉)
School of ICE, Sungkyunkwan University, Jangan-gu, Suwon, Republic of Korea
e-mail: jinsookim@skku.edu

Keywords Virtualization · Virtual machine · Network · Optimization

1 Introduction

By virtualizing existing server nodes in distributed or parallel computing systems and consolidating them into fewer physical nodes, virtualization technology reduces the cost to own the systems, as well as the management cost [26]. Moreover, energy efficiency is greatly improved by reducing the physical nodes. This advantage is increasingly important in the large-scale computing environment due to the rapidly rising energy costs [17]. Therefore, a lot of virtualized parallel and distributed computing platforms [9, 15] have been introduced and continue to become more popular.

To implement virtualization technology, there should be an additional software layer, called a virtual machine monitor or a hypervisor, inserted between the existing operating system and hardware to manage the physical hardware resource and virtual machines. Generally, modern hypervisor implementations are divided into two categories: the host-based approach that uses modified operating systems to provide virtual machine monitoring, and the bare-metal approach that employs small dedicated hypervisors that run on physical machines like operating systems in nonvirtualized environments. FreeBSD jail [14], Linux-VServer [27], OpenVZ [22], and Solaris Zones [23] are examples of the host-based approach, and VMware ESX server [29] and Xen [3] are examples of the bare-metal approach.

Regardless of the type, a hypervisor induces significant overhead that harms the system throughput. However, with the aid of virtualization instruction sets employed in the modern processors along with the effectively designed hypervisors, the performance loss of a processor-bound virtual machine now becomes less than 3%, in comparison with that in the nonvirtualized environment [3]. However, the performance degradation of an I/O-bound virtual machine is still significant because virtualizing I/O devices, such as disks or network interface cards, to be shared by multiple guest systems inherently requires the intervention of the hypervisor. The hypervisor has to control all the I/O requests issued by guest systems in order to prevent the unauthorized access to these physical devices, as well as to arbitrate their use among guest systems.

Among all virtualized devices, the relative overhead of the virtualized network devices is higher than that of other virtual devices because the working cycle of a network operation is usually very short. Therefore, the relative length of a hypervisor intervention, compared to the device working time, is longer for a network operation than the operation for the other types of I/O devices [2, 21, 22].

When each virtual machine has an independent mission, or the interactions between virtual machines occur infrequently, the network virtualization overhead may not become a significant source of performance drawback. However, in distributed parallel computing systems, where each virtual machine works cooperatively for a common goal, network performance critically affects the overall system throughput due to the large overhead induced by frequent networking between virtualized server nodes.

Therefore, there have been a lot of research efforts to reduce the overhead and side effects in virtualizing network devices. Restricting context switches between virtual machines to process network operations [28], devising a novel high-level interface to support virtualized networking efficiently [20], and adding a new I/O channel for network operations [25] are good examples that successfully reduce the networking overhead in virtualized systems.

These existing research results aimed to improve the performance while keeping all the benefits of the virtualization, including isolation and security among virtual machines. However, we could gain greater performance improvement if we sacrifice unnecessary benefits from virtualization in some specific target systems. Every node in distributed systems or parallel computing systems has the same owner and operates for the same goal. Thus, all the virtual machines trust each other and, therefore, the isolation and security are less of a concern than the overall system throughput.

Our research starts from the precise analysis of the overhead of the networking operations in the virtualized environments with various analysis methods, including CPU performance measurement registers. Based on the analysis results, we redesign the networking mechanism to have low overhead while retaining the performance level. Our approach sacrifices the isolation and security among virtual machines to reduce the overhead by using a shared buffer, which removes the data copy operations across virtual machines and a hypervisor in the network operations. We implement the prototype on Xen to evaluate our solution in comparison to an existing network device virtualization approach, which is currently widely used in real-world systems.

The problems we are to tackle in this research are commonly found in most of virtualization platforms including Xen and VMware ESX. In this paper, we selected Xen as our platform for problem analysis and prototype implementation since, as a full-featured virtualization platform, it is being widely used in both industry and academic spheres, and its source code is open to the public.

The rest of this paper is organized as follows. Section 2 presents the background and motivation, including the network device virtualization in the Xen architecture and its performance drawback, along with the related research results to improve the networking performance in virtualized systems. Then Sect. 3 introduces the design of our solution to improve the network performance for virtualized distributed or parallel computing systems, as well as the implementation issues of our approach. Section 4 evaluates the performance-centric characteristics of the prototype implementation in comparison to the original virtualized network device of Xen. The related work is described in Sect. 5. Finally, we present the conclusion of our research in Sect. 6.

2 Background and motivation

2.1 Xen virtual network architecture

Xen [3] is a popular bare-metal virtualization platform, which is now popular in industry as well as in academia. While some other commercial virtualization platforms only support full-virtualization [29], in which the unmodified operating systems run as virtualized guest systems, Xen also employs para-virtualization [3], in which guest operating systems are modified to enhance performance. Due to para-virtualization technology, Xen almost achieves near-native performance, especially for CPU-bound guest systems.

Xen allows only privileged domains[1] to directly access physical devices. In this paper, we call these domains the *Isolated Driver Domains* (IDDs), while domains other than IDDs are denoted simply as just the *guest domains*.

Because most I/O devices are designed to be used by a software entity and managed by a hypervisor in virtualized systems, the guest domains may not be allowed to access them directly. Therefore, the Xen hypervisor provides virtual devices to its virtual machines. A virtual device is apparently a normal device in the viewpoint of each guest system. However, any request to the virtual device from a guest system will be sent to the hypervisor, and the hypervisor will forward it to an IDD that masters the corresponding native device. In contrast, after processing the request, the IDD sends the result back to the guest system, from which the request was originated, through the hypervisor again. By employing this virtual device structure, the Xen hypervisor enables multiple guest domains to share a physical device.

Xen employs the split device driver model to implement the virtual device drivers. A virtual device driver consists of two sub-device drivers called the front-end driver, which is located in the guest domains, and the back-end, which resides in the IDDs. The front-end driver provides interfaces of the virtual device driver to guest domains, and the back-end driver handles all the operations forwarded from the front-end drivers in guest domains and returns the results to the front-end drivers. In short, the back-end driver multiplexes the requests from each guest domain and demultiplexes the results to the requests to send back to each guest domain.

The front-end and back-end drivers communicate with each other using two primitives, shared memory, called an *I/O ring*, and event channels.

An I/O ring is a circular queue to be shared by the IDD and the guest domains. The guest domains put the requests in the I/O ring, and the IDD collects the request by dequeueing it from the I/O ring. In addition, the I/O ring plays the role of a medium for the IDD to transfer the result data to the guest domains.

An I/O ring is established using grant table operations, which are a set of hypercalls[2] to share memory pages between two domains. Suppose that *Domain A* allows *Domain B* to access a memory page it has by presenting the grant reference of the memory page to Xen hypervisor; then, *Domain B* maps the memory page into its address space using grant table operations and accesses the memory page provided by *Domain A*. Once the use of the memory page is finished, the memory page must be explicitly unmapped using grant table operations again.

An event channel is an asynchronous way for a domain to notify an event to other domains. When a domain sends an event through an event channel, the Xen hypervisor marks the corresponding event as pending, and the target domain processes the event when it is scheduled.

As shown in Fig. 1, the original Xen virtual network device driver, *VETH* (virtual ethernet), consists of *Netfront*, the front-end driver for VETH, and *Netback*, the back-end driver. Netfront is located in every guest domain that wants to use VETH, while

[1] Domain denotes virtual machine in Xen terminology.

[2] A domain calls hypercalls to synchronously invoke a service in the Xen hypervisor.

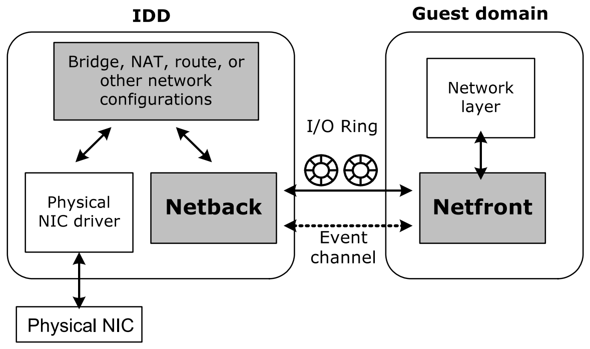

Fig. 1 Simplified current Xen network virtualization

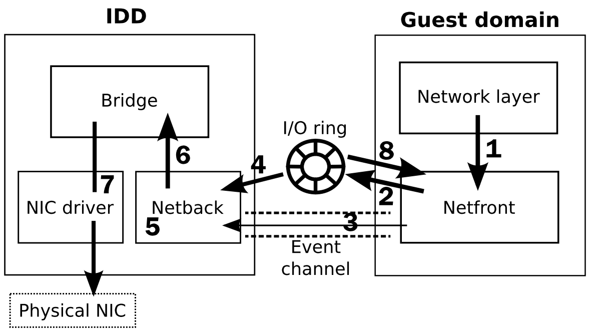

Fig. 2 Data transmission procedure through VETH architecture

the Netback is located in the IDD that actually manages the physical network devices. Since all the data passing through the network devices must flow between Netfront and Netback and they are separately located in different domains, the efficiency of communication mechanism between Netback and Netfront is critical to the system performance.

In detail, a guest domain transmits data through VETH by the following steps as depicted in Fig. 2.

1. (**Data acquisition** ⟨**guest domain**⟩) Network layer of guest domain pushes down a packet to be transmitted to netfront device.
2. (**Data processing** ⟨**guest domain**⟩) Netfront makes a grant reference of the packet, and enqueues a transmit request into the I/O ring, along with control information and the grant reference of the packet.
3. (**Notification** ⟨**guest domain**⟩) Netfront notifies Netback of the new transmit request through event channel.
4. (**Interrupt** ⟨**IDD**⟩) Netback dequeues the transmit request from the I/O ring.
5. (**Data processing** ⟨**IDD**⟩) Netback reads the grant reference of the packet from the transmit request and maps the packet into the address space of the IDD using grant table operations. Then, Netback copies the packet into a new socket buffer to be used by the actual device driver for the physical network device. The mapped packet is unmapped using the grant table operations after finishing the copy. Finally, Netback enqueues the transmit reply that informs the completion of transmitting the packet into the I/O ring.

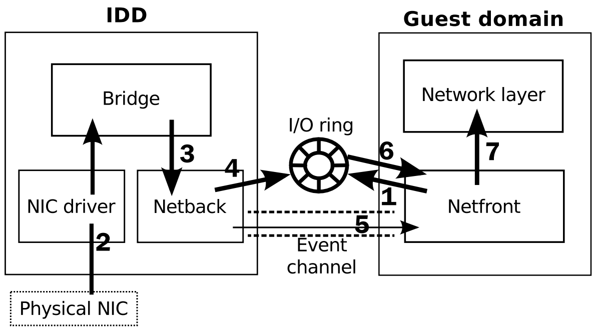

Fig. 3 Data reception procedure through VETH architecture

6. (**Bridging** ⟨**IDD**⟩) The socket buffer which contains the copied packet is pushed into the bridge between Netback and the physical device driver.
7. (**Transmission** ⟨**IDD**⟩) The physical device actually transmits the packet over the physical network link.
8. (**Cleaning up** ⟨**guest domain**⟩) When Netfront sees the transmit reply from the I/O ring, it frees the buffer from the packet.

In contrast, a guest domain receives data from VETH by the following steps as shown in Fig. 3.

1. (**Preprocessing** ⟨**guest domain**⟩) Netfront allocates buffers for incoming packets and makes grant references of the buffers. Then Netfront enqueues the receive requests into the I/O ring. Each receive request contains the control information and grant reference of the buffer.
2. (**Data acquisition** ⟨**IDD**⟩) The physical network device receives a packet and pushes it into the bridge.
3. (**Bridging** ⟨**IDD**⟩) The bridge passes the packet over to Netback.
4. (**Data processing** ⟨**IDD**⟩) Netback sets up the buffers. Netback reads the grant reference of the allocated buffer from the receive request and maps the buffer into the address space of IDD. Then, Netback copies the packet into the buffer. Finally, Netback enqueues the receive reply that specifies the new packet arrival into the I/O ring.
5. (**Notification** ⟨**IDD**⟩) Netback notifies Netfront of the receive reply through the event channel.
6. (**Interrupt** ⟨**guest domain**⟩) Netfront dequeues the receive reply from the I/O ring.
7. (**Data processing** ⟨**guest domain**⟩) Netfront pushes the received packet up to the network layer.

2.2 Motivation

In order to identify the source of the overhead in networking, we observe the processor usage of the IDD with Xenoprof [21], while a guest domain transmits and receives 64 Kbytes packets over the TCP connection, respectively, for 300 seconds. The measurement results are shown in Table 1, with the kernel symbols frequently invoked in the IDD, their execution times, and the L2 cache miss rate.

Table 1 Some symbols frequently used in network processing in IDD when guest domain executes network operations (sorted by % CPU)

Symbol name	% CPU	% Cache miss
Bulk data transmitting		
`do_grant_table_op`	5.33	7.09
`tg3_poll`	5.14	15.1
`tg3_start_xmit_dma_bug`	3.91	0.43
`net_tx_action`	3.64	5.17
`hypercall`	3.35	0.04
`hypercall_page`	3.17	0.03
`skb_segment`	3.04	3.31
`_spin_lock_irqsave`	1.64	0.37
Bulk data receiving		
`gnttab_copy`	8.98	25.18
`tg3_poll`	3.48	4.05
`do_grant_table_op`	3.37	4.29
`net_tx_action`	2.92	5.06
`hypercall`	2.70	0.13
`hypercall_page`	2.58	0.25
`net_rx_action`	2.33	1.34
`_spin_lock_irqsave`	2.00	1.17

The most frequently executed functions while both transmitting and receiving data are the grant table operations, such as `do_grant_table_op` and `gnttab_copy`. Grant table operations are used to map the buffer in guest domains into the address space of the IDD, as described in Sect. 2.1.

Every transmission and receipt of a single packet involves at least two grant table operations, mapping and unmapping. Grant table operations are a kind of hypercalls, and, naturally, frequent invocation of hypercall degrades the overall performance of the system because calling a hypercall induces the privilege level switchings. To optimize this process, Xen could merge several hypercalls, including grant table operations, into a single hypercall. In spite of this optimization, we observe that grant table operations are still extensively used, as shown in Table 1.

Copying packets increases memory footprints, and thus increases L2 cache misses. Copying bulk data in receiving operations shows noteworthy CPU utilization, as well as high L2 cache misses in Table 1. Copying operations during receiving operations degrade the performance even more than transmitting operations. Copying the received packet into the buffer in the guest domain incurs many L2 cache misses because NIC uploads the packet into the main memory using DMA, and the packet is not cached in the L2 cache located inside the CPU.

Therefore, in order to effectively reduce CPU consumption and L2 cache misses, both of which are significant sources of performance decline, the use of grant table operations and copying packets have to be avoided in critical communication paths.

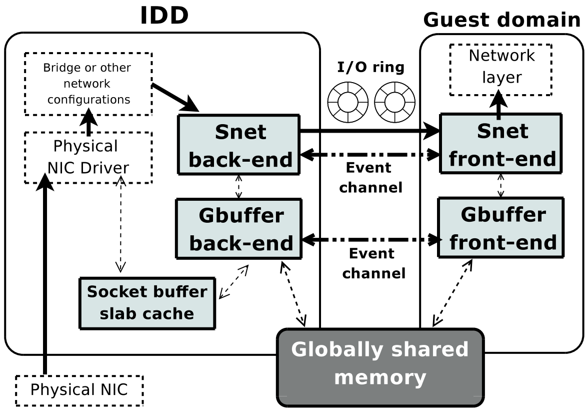

Fig. 4 Overall architecture of SETH network virtualization

3 SETH: our approach

3.1 Design

The ultimate design principle of **SETH** (Simply-shared Ethernet) is to minimize the use of grant table operations and to avoid copying packets across domains while networking.

While transmitting a packet, it is inefficient to map and copy the entire packet in the guest domain into the IDD. Thus, SETH does not map and copy the entire packet. Instead, SETH exploits only the necessary information from the packet in order to make a socket buffer used in the IDD and to bridge the packet, similar to the approach used by Menon et al.

When receiving a packet, it is inevitable to conduct mapping and copying the received packet, since the destination of the packet is the guest domain, and thus the IDD must hand the packet to the recipient domain. To avoid mapping and copying, SETH makes all the socket buffers in the IDD reside in the preallocated shared memory, while VETH dynamically shares the socket buffer, maps the buffer, and copies the packet into the buffer of the guest domain. Thus every received packet is accessible from other domains, and, naturally, copying and mapping are not necessary.

Figure 4 shows the overall architecture of SETH. We divide the tasks of designing SETH into three subtasks: handling network operations, managing preallocated shared memory, and making a network stack in the IDD use the preallocated shared memory.

Three components of SETH are responsible for these three tasks: *snet*, *gbuffer*, and the *socket buffer slab cache*.

Snet has two components: *snet front-end* and *snet back-end*. Each of them handles network operations similar to Netfront and Netback in the VETH architecture, respectively. The Snet front-end works as a network device driver interface of the snet network device driver in guest domains, and the Snet back-end connects the Snet front-end with the physical network device driver. As the back-end driver in VETH, the Snet back-end directly interfaces with physical network device driver in the operating system of the IDD.

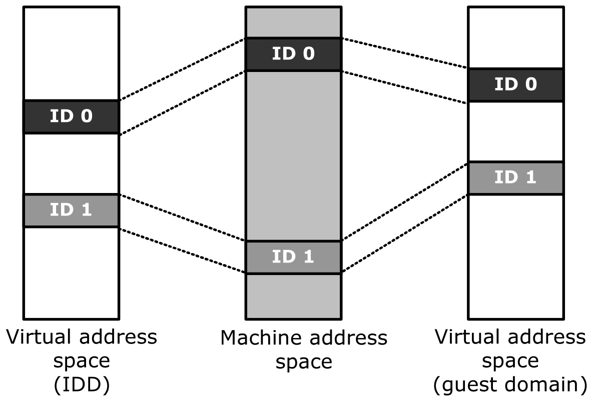
ID 0
ID 1
ID 0
ID 1
ID 0
ID 1
Virtual address space (IDD)
Machine address space
Virtual address space (guest domain)

Fig. 5 Creating preallocated shared memory

Gbuffer initializes and manages preallocated shared memory, which is to be shared among domains to pass the network packets. Gbuffer is also implemented into two separate parts: *gbuffer front-end*, which resides in the guest domains, and *gbuffer back-end* in the IDD.

The **socket buffer slab cache** makes the network stack in the IDD use data buffers for packets, which are located in the preallocated shared memory. Note that gbuffer and the socket buffer slab cache are made in order to remove the dynamic invocation of grant table operations in the receiving path; they are not involved in the transmitting path.

The following subsections describe gbuffer, the socket buffer slab cache, and snet in detail, especially focusing on the interaction between these modules.

3.2 Gbuffer and socket buffer slab cache

Gbuffer prepares preallocated shared memory before the snet network device driver begins to process network operations. SETH does not use the grant table operations in creating preallocated shared memory. Instead, it acquires 4 MBytes of contiguous memory from the domain heap and maps it into domains, as shown in Fig. 5.

The newly introduced hypercall in SETH allocates contiguous 1,024 pages from the domain heap and returns the starting machine address of that area. Domains map these pages into an empty contiguous virtual memory area, which is acquired from the balloon driver of Xen. Each area has a unique zone id, and it is increased by one whenever a new 4 Mbytes of shared memory is preallocated.

Preallocated shared memory is used as a buffer in the IDD for receiving packets. Whenever gbuffer back-end creates a preallocated shared memory, it is notified to gbuffer front-end through the event channel. In addition, gbuffer front-end maps the preallocated shared memory into the virtual address space of the guest domain and gives that area the same zone id to that in IDD. Thus, with the zone id and offset, each domain accesses the same data without dynamically sharing the memory pages, using grant table operations.

In addition, gbuffer back-end works as a kind of memory allocator, based on the preallocated shared memory. It provides page-sized memory from the preallocated shared memory, like the buddy allocator in the Linux kernel. When the gbuffer back-end allocates all the pages in a zone, it creates additional preallocated shared memory as mentioned before.

We modify the socket buffer slab cache in SETH to work on top of gbuffer back-end, whereas all the slab caches run on top of the buddy allocator. The socket buffer slab cache acquires pages from and releases pages to gbuffer back-end. We also modify APIs, such as `alloc_skb()`, which allocate socket buffers to use our socket buffer slab cache when allocating the data buffers for packets.

However, SETH retains the way of allocating the socket buffer structure. The socket buffer structure is allocated from normal memory area not in preallocated shared memory. Thus, every data buffer of the socket buffer in IDD is from the pre-allocated shared memory and the guest domain accesses it without using grant table operations, once the guest domain maps the preallocated shared memory in its address space. In this mechanism, all the received packets from the physical network device driver in the IDD are located in the preallocated shared memory, and the guest domain accesses the received packet with the zone id and offset.

Gbuffer provides helper APIs as follows.

– `custom_kmem_getpages()` is a custom version of `kmem_getpages()` of the original virtualized Linux kernel. It is used to allocate page frames from the free page pool in the virtual machine. Like `kmem_getpages()`, the custom version provides page frames. However, it gets the free page frames from the preallocated shared memory.
– `custom_kmem_freepages()` is a custom version of `kmem_freepages()`. It releases pages, which were allocated from `custom_kmem_getpages()`, to the preallocated shared memory.
– `custom_get_base()` receives a virtual address and tells the preallocated shared memory zone id where the virtual address is from and the starting virtual address of the zone. When the virtual address is not included in the preallocated shared memory, this API returns NULL, which can be used to test whether an address is included in the preallocated shared memory.

The preallocated shared memory consumes 4 Mbytes of contiguous virtual address space per shared memory zone. According to our observation in receiving bulk data, a small preallocated shared memory is enough in network processing. Only two preallocated shared memory zones (8 Mbytes) are observed to be utilized in receiving bulk data with one NIC in the preliminary evaluation of the prototype.

3.3 Snet

This section explains the mechanisms of the snet operations: transmitting data, receiving data, and interfacing with physical network device driver. We describe implementation details focusing on the interactions between the snet back-end and front-end device driver.

3.3.1 Transmitting packets

In VETH, when transmitting a packet in guest domain, the packet may be accessed in the following three cases. First, Netback reads the entire packet. Netback copies the

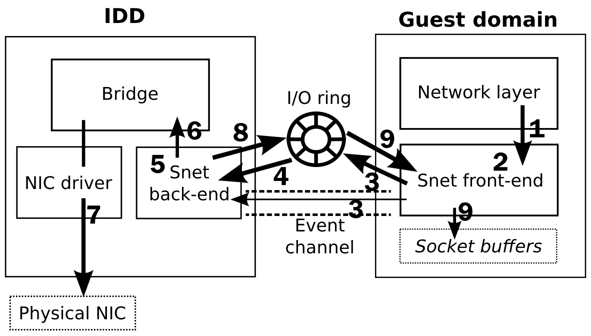

Fig. 6 Data transmission procedure through SETH architecture

entire packet to the data buffer of the newly-created socket buffer. Second, the bridge reads the packet for bridging. Finally, the physical network device reads the entire packet to download onto its onboard memory to transmit over the link. Note that the physical network device needs only the starting machine address and the size of the packet, since the physical network device usually does not utilize MMU in accessing memory.

Snet removes the first two steps done in VETH when transmitting packets. Since snet back-end directly interfaces with the physical network device, snet back-end does not access any single byte in the packet for bridging and thus does not need to copy the entire packet. Snet back-end makes a new socket buffer structure, which handles the packet, and then the physical network device driver transmits the packet using the starting machine address of the packet specified in the transmit request.

Snet back-end creates both a pseudo packet and socket buffer structure, since the bridge of the IDD needs to read the header of the packet. The header of the pseudo packet is synthesized from the necessary information, such as the destination MAC address specified in the transmit request, and the payload of the pseudo packet is empty since the physical network device driver directly uses the machine address of the actual packet.

The following illustrates the steps for transmitting packets in snet as shown in Fig. 6:

1. (**Data acquisition** ⟨**guest domain**⟩) Network layer pushes down a packet to be transmitted into snet front-end.
2. (**Control information construction** ⟨**guest domain**⟩) Snet front-end creates a transmit request. A transmit request contains all the necessary information that is used in the IDD to create a new socket buffer structure that handles the packet in the IDD. It includes the starting machine address of the packet, size of the packet, offset of the MAC header, and so on. In SETH, a transmit request additionally contains the MAC address. Along with this, Snet front-end increases the usage count of the socket buffer by one, since it has to be free only after the physical network device driver successfully transmits the packet.
3. (**Notification** ⟨**guest domain**⟩) Snet front-end enqueues the transmit request into the I/O ring and notifies snet back-end.
4. (**Interrupt** ⟨**IDD**⟩) Snet back-end dequeues the transmit request from the I/O ring.

5. (**Socket buffer re-organization** ⟨**IDD**⟩) Snet back-end creates a new socket buffer structure, which handles the packet as follows. First, snet back-end allocates a new socket buffer and releases the data buffer of the socket buffer. Then snet back-end updates the socket buffer structure with the information specified in the transmit request. For example, the head in the socket buffer structure points to the starting machine address of the packet, and other layout fields, including data, tail, and end, are updated based on the starting machine address of the packet. In the bridge-enabled mode of SETH, Snet back-end does not release the data buffer of the socket buffer and reconstructs pseudo-header with the necessary information, including the destination MAC address of the actual packet. This pseudo-header is used in bridging.
6. (**Bridging** ⟨**IDD**⟩) Snet back-end directly interfaces with the physical network device driver or uses the bridge, alternatively. When directly interfacing with the physical network device driver, SETH back-end invokes `hard_start_xmit()` virtual function of the physical network device driver. When using the bridge, SETH back-end hands over the socket buffer to the bridge, which enqueues the socket buffer into the send queue of the physical network device driver.
7. (**Transmission** ⟨**IDD**⟩) The socket buffer newly-created in IDD contains the starting machine address and the size of the packet. Using this information, the physical network device driver orders the network device to download the packet through DMA, and then the network device actually transmits the packet over the physical network link.
8. (**Notification** ⟨**IDD**⟩) After the physical network device driver is notified of the successful transmission, snet back-end sends a transmit reply to snet front-end.
9. (**Cleaning up** ⟨**guest domain**⟩) Once snet front-end receives the transmit reply, the socket buffer that contains the transmitted packet is freed.

3.3.2 Receiving packets

In both cases of VETH and SETH, accessing the received packets in the IDD is inevitable, since the physical network device driver automatically uploads the received packet from its on-board memory into the memory of the IDD, contrary to transmitting packets. Figure 7 shows how the guest domain accesses the received packet both in VETH and SETH.

In VETH, the received packet is located in the data buffer of a socket buffer in IDD. Since the guest domain cannot access the data buffer in other domains, VETH copies the packet into the shared buffer through grant table operations, as shown in Fig. 7(a). Then the guest domain utilizes the received packet. During this mechanism, every receiving operation involves several grant table operations, such as mapping, copying, and unmapping.

SETH allocates every data buffer of a socket buffer used in IDD in the preallocated shared memory, as shown in Fig. 7(b), due to gbuffer and socket buffer slab cache explained in the previous section. Thus, snet does not exploit grant table operations in every receiving operation, but directly accesses the packet.

Following illustrates every step involved in receiving packets in snet as illustrated in Fig. 8.

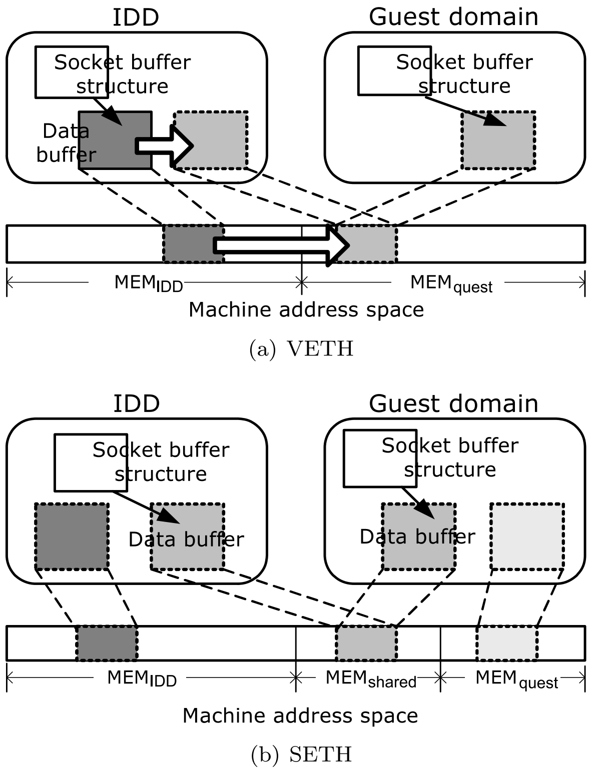

Fig. 7 Access data buffer of the socket buffer in guest domain

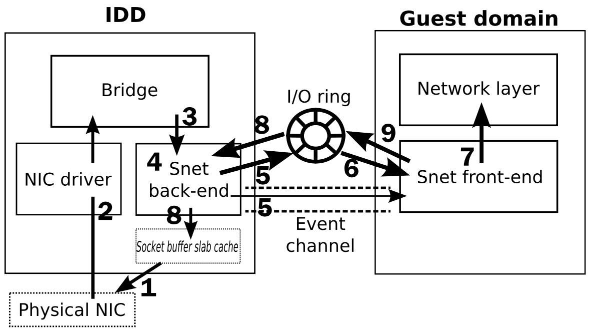

Fig. 8 Data reception procedure through SETH architecture

1. (**Buffer provisioning** ⟨**IDD**⟩) The physical network device driver prepares socket buffers for incoming packets. The data buffers of the socket buffers are already allocated from the preallocated shared memory, due to gbuffer and the socket buffer slab cache, before the physical network device uploads the received packet into the memory of IDD.
2. (**Data acquisition** ⟨**IDD**⟩) When a packet arrives, the physical network device driver passes over the socket buffer, which contains the received packet to the upper network layer.

3. (**Bridging** ⟨**IDD**⟩) We slightly modify the network layer to directly push the socket buffer into the receive queue of snet back-end. Alternatively, when using the bridge-enabled mode of SETH, the network layer pushes the received data to the bridge. The bridge enqueues the socket buffer into the receive queue of snet back-end.
4. (**Control information construction** ⟨**IDD**⟩) Snet back-end makes a receive request. A receive request contains the shared memory zone id, offset of the starting address of the packet in the shared memory zone, size of the packet, and other control information necessary to reorganize the socket buffer in the guest domain. The shared memory zone id can be acquired from `custom_get_base()`, provided by gbuffer, by presenting the virtual address of the packet. Additionally, snet back-end increases the usage counter of the socket buffer by one, since the socket buffer should not be freed until the guest domain frees the data buffer of the socket buffer.
5. (**Notification** ⟨**IDD**⟩) Snet back-end pushes the receive request into the I/O ring and notifies snet front-end.
6. (**Interrupt** ⟨**guest domain**⟩) Snet front-end dequeues the receive request from the I/O ring.
7. (**Socket buffer reorganization** ⟨**guest domain**⟩) Snet front-end creates a new socket buffer and frees the data buffer of the socket buffer. Snet front-end calculates the virtual address of the packet from the shared memory zone id and the offset specified in the receive request. Then snet front-end reorganizes the socket buffer structure with the starting address of the packet and the other control information in the receive request. Snet front-end passes the packet up to the network layer.
8. (**Notification** ⟨**guest domain**⟩) After the kernel copies the payload data of the packet to user process, the socket buffer is usually freed. When the data buffer is freed in the guest domain, snet front-end creates a receive reply and sends it to snet back-end.
9. (**Cleaning up** ⟨**IDD**⟩) Once snet back-end receives the receive reply, snet back-end decreases the usage count of the data buffer. If the usage count is zero, the data buffer is freed to the socket buffer slab cache and recycled.

3.3.3 Interfacing with physical NIC driver

Virtual network in Xen allows various network configurations, from simple to complex. For example, administrators can use the bridge netfilter, NAT, routing, and so on. Along with the flexibility Xen provides, this adds more computational overhead to amend each packet to fit to the network configuration.

Snet back-end directly interfaces with the physical NIC drivers using kernel patches, which insert several hooks in the network layer. When transmitting data, snet back-end directly invokes `hard_start_xmit()` virtual function of the outgoing physical network device. When receiving data, the modified `netif_receive_skb()` of network layer directly enqueues the received socket buffer into the receive queue of snet back-end. Optionally, snet back-end can be configured to use the bridge or other network configurations. It processes the packet before calling the physical device driver functions, as Netback in VETH does.

4 Evaluation

4.1 Evaluation environment

To evaluate the improvement of network performance and the reduction of processor overhead, we implement the prototype of our approach. The prototype is implemented on the Xen-3.2.1 hypervisor, with paravirtualized Linux 2.6.18. We use the unmodified Linux kernel with the same release number as the native counterpart to be compared in our evaluation. The host system is equipped with Intel Pentium D 930, a dual-core processor at 3.0 GHz, and 2 GB main memory, along with a Broadcom BCM5701 gigabit ethernet NIC. The network performance as well as the overhead of SETH, is compared with VETH in the same software configuration and also the native Linux kernel.

Our evaluation was done with synthetic benchmarks, which conduct the same operations continually, and real-world workloads. Whereas the synthetic benchmarks present microscopic analysis of the networking overhead, the real-world workload evaluation provides intuition about the performance enhancement by employing SETH at the whole system level.

All the benchmarks and workloads evaluated in this section use TCP as the transport protocol. Although our prototype implementation is applicable to any transport protocols such as TCP, UDP, and RTP, we primarily target the workloads that use TCP protocol. The main reason is that most of the enterprise-scale and high-performance computing applications rely on the TCP protocol for data transfer since these workloads require high reliability and efficiency.

The choice of the virtual machine scheduler affects the scheduling latency. In our evaluation, we use the credit scheduler, the default virtual machine scheduler with its default configurations. In addition, we do not set processor-affinity for guest systems, so that they can run on any processor to obtain the response as fast as possible.

4.2 Microscopic benchmarks

To compare the processor overhead for handling networking requests, we define a new performance metric, **Transfer Cost**, because the overall performance of benchmark program reflects the networking overhead not accurately, but approximately.

Transfer cost (TC) is calculated as

$$\mathsf{TC} = \frac{\textit{Million Clock Ticks}}{\textit{Transferred Data Size in Bytes}}.$$

TC means how many processor cycles are required to transmit or receive data of the unit size. The number of processor cycles during a network operation is sampled by *Xenoprof*, a performance profiler for Xen. When computing-intensive virtual machines run with the networking virtual machines, all processor cycles are utilized in a work-conserving manner, and no idle cycles remain. Therefore, we can tell that a networking mechanism with lower TC incurs less performance drop of the computing-intensive virtual machines as well as the networking virtual machine than that with higher TC.

Table 2 Round-trip time and CPU utilization for a 4-byte packet ping-pong test

	Round-trip time (ms.)	CPU utilization (%)
native	125.35	14.14
VETH	127.84	44.16
SETH	126.59	36.77

4.2.1 Round-trip time

Table 2 shows round-trip time and CPU utilization for a 4-byte ping-pong experiment. Another computer, which has the same hardware configuration and connects to the same subnet, is used for the experiment.

A round-trip time of a packet is an elapsed time that aggregates the time required for sending a packet and receiving the reply packet from the counterpart. Since the size of the ping packet is small, the round-trip time depends not on the data copy or network transfer time, but on the sum of initializing cost for every step in processing packets. Therefore, the latency of the ping-pong trials implies more than the bandwidth.

According to Table 2, the round-trip times of each method do not show significant differences, while the average processor utilization of VETH and SETH records 1.89 and 3.12 times higher numbers than that of native, respectively. This indicates that the processor performance or the amount of processor resource is not a critical factor in determining the round-trip time. In other words, the latency is affected by the network delay, rather than the network processing overhead.

However, if there is a high demand on the processor resource by some other tasks during the ping-pong task, their performance is negatively impacted by the high processor utilization. Even though the processor utilization of a network mechanism has a negligible effect on the network latency, it directly relates to the throughput of other processor-bound tasks. Therefore, we can tell that by using SETH, we can significantly reduce the side-effects to the throughput of other processor-bound tasks. Especially when the size of network packets is small, like in the ping-pong experiments, we expect approximately 17% performance improvement, based on the evaluation results.

4.2.2 Transmitting

To identify the transmission performance as well as its overhead, we measure the transmission bandwidth, while changing the transmitting data size. The sent packet is received by a host using the native Linux kernel.

As shown in Fig. 9, there are little differences in the transmission bandwidth among all three architectures. This result is consistent with the ping-pong experiment results. When the packet size is small, the time needed to copy data to the buffer is relatively short in the VETH architecture. Therefore, the elapsed time in kernel to process a packet is barely different from that in the native Linux kernel, and it results in the negligible degradation of the transmission bandwidth.

However, as the data size to transmit grows, the processor utilization increases, which results in the degradation of transmission throughput. It is verified that SETH

Fig. 9 Transmitting bulk data in various sizes

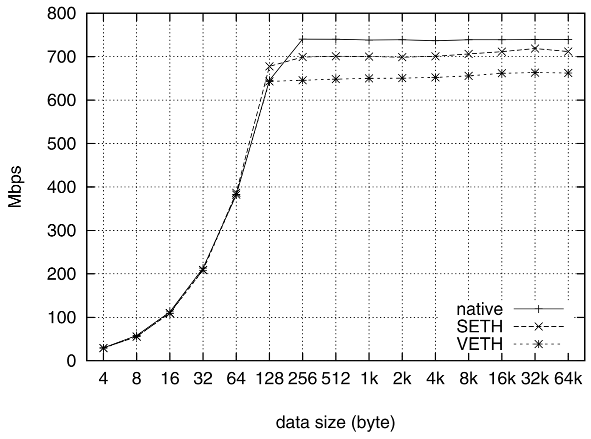

Table 3 CPU utilization and transfer cost (TC) measured in transmitting 64 Kbyte data

Approach	CPU utilization (%)	Transfer cost (TC)	Normalized transfer cost
native	30.14	9.991	1.000
SETH	73.04	24.170	2.420
VETH	94.51	33.457	3.349

outperforms VETH by over 50 Mbps, when the transmission of the unit size larger than 128 KB, and that the performance of SETH is closer to native, rather than VETH. This result supports our hypothesis that the data copy operation to the device driver buffer across the domains is one of the major causes of the networking overhead.

The process utilization during 64 KB-unit transmission, when all three approaches perform similarly, is shown in Table 3. Although all of the approaches perform similar throughputs, VETH consumes 3.3 times more processor resources than the native architecture, while SETH consumes 2.4 times more. This indicates that SETH significantly reduces the processor resource usage in comparison to VETH.

4.2.3 Receiving

Figure 10 shows the receiving bandwidth of native, VETH, and SETH from 4-byte to 64-Kbyte data. Similar to the round-trip time results in the previous section, receiving bandwidth among the network architectures does not differ considerably. The bandwidth of the unvirtualized Linux is saturated at 256 bytes for 740.78 Mbps and sustains its bandwidth as the data size increases.

VETH shows the maximum bandwidth at 256 bytes for 728.84 Mbps; we see the small bandwidth fluctuation after 256 bytes, and the bandwidth is gradually decreased. The maximum observed bandwidth of SETH is 739.923 Mbps at 256 bytes. The bandwidth of SETH is slightly reduced between 512 bytes and 1 Kbytes. One of the reasons for the decreased bandwidth is that snet back-end occasionally cannot enqueue new receive requests into the I/O ring, since the I/O ring is full with the previous receive requests.

Fig. 10 Receiving bulk data in various sizes

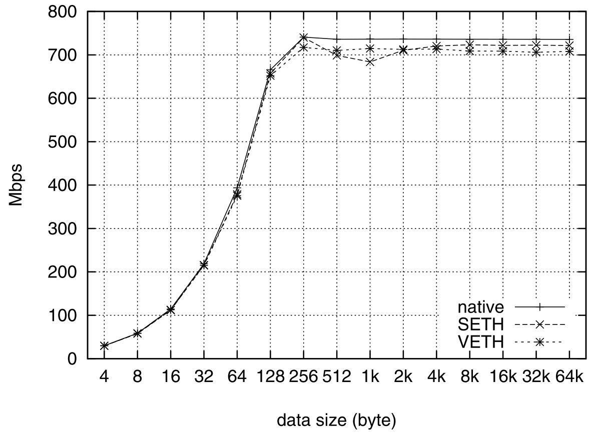

Table 4 CPU utilization and transfer cost (TC) measured in receiving 2-Kbyte packets

Approach	CPU utilization (%)	Transfer cost (TC)	Normalized transfer cost
native	62.34	19.355	1.00
SETH	115.13	35.527	1.836
VETH	144.47	46.732	2.414

In fact, when the guest domain receives 512-byte and 1-Kbyte data, we discover that the I/O ring is sometimes full of the receive requests, and the measured bandwidth at those data sizes is a little smaller than VETH. While the I/O ring is the same size for both VETH and SETH, the size of a receive request in SETH is larger than that in VETH. Thus, the I/O ring of SETH can be more easily filled than that of VETH.

Table 4 shows the CPU utilization and transfer cost measured when a guest domain receives 1500-byte packets. Native displays the smallest CPU consumption and transfer cost. Similar to the previous round-trip time experiment, VETH and SETH consumes higher CPU time to process receiving packets, compared to native.

For the detailed overhead analysis, we profile the whole system, including the IDD and the guest domain, when the guest domain receives 1500-byte packets for 300 seconds, both in VETH and SETH. We group the entire sampling results of CPU utilization and L2 cache misses into five categories and present them in Table 5. In the table, `tg3` denotes the device driver of our physical network device. Note that when we run the same experiment in native, we observe 239 million L2 cache misses with 62.34% CPU utilization.

SETH shows smaller CPU utilization and L2 cache misses compared to VETH. CPU utilization of SETH is roughly 80% of that of VETH, and L2 cache misses of SETH is about 64% of those of VETH. Increased CPU utilization and L2 cache misses in VETH mainly come from two copy operations:

(1) a grant copy, which copies the received packet in the IDD to the kernel space of the guest domain, and

Table 5 CPU utilization, cache misses, and cache miss ratio in several components when receiving 2-Kbyte packets

	VETH			SETH		
	% CPU	Cache miss		% CPU	Cache miss	
Total	144.466	1007.8	(100.00)	115.131	642.3	(100.00)
	Xen hypervisor					
Xen kernel	38.730	470.8	(46.717)	20.647	159.8	(24.881)
scheduling & context switching	6.307	87.0	(8.63)	5.731	73.2	(11.403)
grant table operations	15.743	291.5	(28.92)	0.021	0.1	(0.02)
	IDD					
Linux kernel	44.081	194.0	(19.25)	38.620	166.5	(25.92)
spin lock	5.561	11.4	(1.14)	6.446	28.7	(4.47)
hypercall	2.839	1.3	(0.13)	2.389	0.2	(0.02)
network management	9.306	68.9	(6.84)	9.163	74.8	(11.64)
bridge	9.678	13.8	(1.37)	7.891	7.5	(1.18)
Modules	12.987	114.6	(11.37)	8.807	83.6	(13.01)
tg3	5.001	41.9	(4.16)	4.129	43.9	(6.84)
gbuffer back-end	–	–	(–)	0.260	0.2	(0.03)
snet back-end	–	–	(–)	4.418	39.5	(6.15)
netback	7.986	72.7	(7.21)	–	–	(–)
	Guest domain					
Linux kernel	40.593	145.9	(14.47)	40.591	202.6	(31.54)
copy_to_user	4.557	61.6	(6.11)	4.094	100.9	(15.71)
spin lock	6.010	4.2	(0.42)	6.681	9.0	(1.40)
hypercall	1.595	0.1	(0.01)	1.865	0.1	(0.02)
tcp	5.164	3.3	(0.33)	5.367	4.0	(0.62)
ip	1.828	3.3	(0.33)	2.011	10.8	(1.69)
network management	2.713	6.2	(0.62)	3.260	44.4	(6.91)
Modules	6.149	80.7	(8.01)	4.493	28.1	(4.38)
snet front-end	–	–	(–)	4.218	28.1	(4.37)
gbuffer front-end	–	–	(–)	0.275	0.0	(0.01)
netfront	6.149	80.7	(8.01)	–	–	(–)

(2) a copy to user, which copies the payload data of the copied packet in the previous step to the user space of the guest domain.

These two copy operations not only considerably consume CPU time, but also incur many L2 cache misses. Approximately half of the L2 cache misses in the grant copy operations are cold misses in reading the received packets, since the physical network device uploads the packets to the memory of the IDD using DMA. The other half is write misses in writing the read packets to the memory of the guest domain.

These write misses are inevitable because the target buffers for grant copy operations are changed continuously. A copy to user operations also causes considerable L2 cache misses. Copy to user operations does not cause write misses, since the user process in the experiment uses the same buffer for receiving data. L2 cache misses of VETH are smaller than those of SETH, since some parts of the packets are cached from the handling of the write misses in the prior grant copy operations.

SETH does not copy the received packets twice, as explained in the previous sections. Removing packet copying during data transfer drastically decreases CPU utilization and L2 cache misses in the grant table operations shown in the table. Unlike the grant copy operations, the copy to user operations in SETH, however, do not take advantage of caching effect like VETH and suffer from cold misses, since the copy to user operations are the first to touch the whole packet. Nonetheless, among the L2 cache misses that occur in data transfer, L2 cache misses in SETH are only 28.6% of those in VETH.

Removing packet copying between the IDD and the guest domain in SETH not only decreases CPU utilization and L2 cache misses in data transfer, but also reduces the entire memory footprint of network processing. Due to the reduced memory footprint in network processing, user processes or other kernel tasks are less affected by the cache pollution. In Table 5, we observe that most of the measurement categories of SETH show smaller CPU utilization, which mainly stems from the decreased L2 cache misses. In particular, the IDD of SETH uses only 83.1% of CPU utilization consumed in the IDD of VETH.

Gbuffer efficiently manages the globally shared memory in receiving bulk data. It consumes only 0.2% of CPU time and shows less than 0.04% of L2 cache misses among entire L2 cache misses. Whereas the network stack frequently allocates and releases data buffers of socket buffers, the socket buffer slab cache rarely allocates memory pages from and releases memory pages to the globally shared memory, since the socket buffer slab cache usually recycles the memory pages. Thus, invoking services in gbuffer is substantially reduced, and we observe small CPU consumption and L2 cache misses in gbuffer.

In both SETH and VETH, CPU spends more than 12% of its time executing spin lock, which shows that network processing is still largely affected by SMP kernel. For SETH, spin lock overhead is a little higher than VETH, since we currently do not implement per-CPU data structures for fast prototyping of SETH.

4.3 Real-world workloads

Table 6 describes the combinations of some real-world workloads that are used in our evaluation. By consolidating various CPU intensive workloads as well as I/O intensive workloads, our experiments are designed to reflect the complex consolidation cases of heterogeneous services [26], which are usually found in both industry and academic spheres.

RayTracing is a synthetic distributed benchmark that consists of a few generic ray tracing simulations. Here, parallel computing nodes are virtualized and run in separate physical servers. Although it is a parallel workload, the communication between the nodes occurs infrequently because the data processing algorithm is well-partitioned. Therefore, in the next experiment, we also consolidate another virtual machine that runs FTP service in order to give greater networking load.

Sysbench OLTP is a *MySQL* server benchmark that handles SQL queries coming from multiple clients over the network. The queries of OLTP are designed to simulate

Table 6 Description of real-world workloads used in our evaluation

Name	Description
RayTracing	2-way parallel distributed ray tracing simulation (The nodes are located at physically separated servers.)
RayTracing + FTPServ.	Consolidation of a RayTracing node and a FTP server in a physical server
Sysbench OLTP [18]	A virtualized MySQL server
RUBIS [1] + SPECCPU [13]	Consolidation of a database server, a web server and an intensive computing server

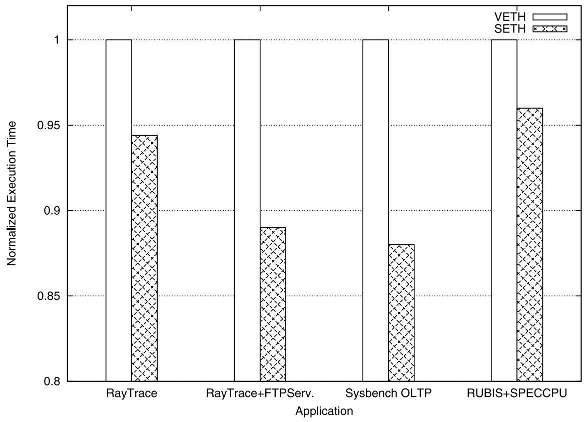

Fig. 11 Normalized execution time of real-world benchmarks (Shorter is better)

financial services. In order to process massive queries, the database server virtual machine utilizes multiple resources such as processor, disk I/O, and network I/O, continuously. This benchmark is chosen to show the effectiveness of SETH under a workload that is both computing- and networking-intensive.

Finally, RUBIS + SPECCPU is an experiment to show the effectiveness of SETH under even more complicated consolidation cases. RUBIS simulates typical 3-tier e-commerce services, which consist of web server, content processor and database server. Both the web server and content processor are run in a virtual machine together hosted in a physical server, and the database server works in a virtual machine hosted in another physical server. The tiers communicate with each other only using TCP connections. SPEC CPU2006, which simultaneously runs with RUBIS, is a set of diverse compute-intensive programs. It is used for measuring the performance impact with the networking overhead of RUBIS. The RUBIS + SPECCPU experiment, thus, uses the execution time of SPEC CPU2006 as its performance metric.

The evaluation results are illustrated in Fig. 11. SETH improves the performance of every workload by 4% to 12%. Especially, as easily expected, the improvement is greater when communication occurs more frequently and the size of packet is smaller.

The OLTP benchmark shows the best performance enhancement. The size of both request and response packets is usually small; the pattern of request and response repeats massively and concurrently. Therefore, the benchmark induces a lot of context switches between the benchmark virtual machine and the IDD and data copies across these two domains, which brings significant performance drop. By reducing the overhead, SETH improves the throughput of the MySQL server by up to 12%. Similarly, RayTracing + FTPserver also obtains about 11% performance enhancements by using SETH.

In RayTracing, the data transfer between nodes is done only twice; at the data partitioning stage and the result aggregation stage. Since the amount of data to transfer is proportional to the amount of data to process, the time for the data transfer takes a significant part in the overall execution time. As a result, SETH shortens the execution time by 5.6%.

Finally and surprisingly, SETH shows the least performance improvement with RUBIS + SPECCPU. RUBIS generates lots of processor demand as well as networking and disk I/O operations since processing a request of RUBIS involves dynamic web page generating operations as well as massive database update operations, both of which demand lots of processor cycles. Therefore, the time for networking in RUBIS is relatively small, and the reduced network overhead in SETH does not greatly decrease the overall execution time. Moreover, the applications of SPEC CPU2006 generally utilize processor cache aggressively. Therefore, the reduced cache miss from SETH might not affect the cache hit ratio of SPEC CPU2006, and this is another major reason for the small performance improvement of RUBIS + SPECCPU.

5 Related work

Many research efforts have been introduced to reduce the overhead of the networking in virtualized environments.

Menon et al. [20] improved Xen virtual network performance by optimizing I/O channel and virtual memory. They also added new capabilities, such as scatter-gather I/O, TCP/IP checksum offload and TCP segmentation offload to the Xen virtual network device, which improve the performance dramatically if the physical NIC supports them. Although they reduce the number of grant table operation invocations, they still depend on the use of grant table operations to exchange packets between domains, and thus suffer from various overheads identified in this paper.

Sugerman et al. [28] optimized the virtual network architecture of *VMware Workstation* by reducing the number of virtual machine switching as much as possible. In addition, Govindan et al. are focusing on the network performance degradation that stems from virtual machine scheduling [12]. In the split device driver model, transferring packets needs context switching between driver domains and guest domains. They modified SEDF virtual machine scheduler to reduce scheduling-induced delays, which also improves the throughput of servers and reduces service latency. Our approach does not deal with the network performance degradation from virtual machine scheduling, but their approaches can be combined with our approach to improve network performance.

Along with the previous approaches that attempt to improve the performance over traditional Ethernet devices, some researches try to use smart network devices. Liu et al. proposed VMM-bypass I/O for Xen, an extension of OS-bypass I/O that was originated from user-level communication, and implemented it on top of InfiniBand network device [19]. User applications of guest domains can access the InfiniBand network device directly without the intervention of VMM or IDDs. VMM-bypass I/O does not dedicate a physical network device to a specific guest virtual machine, but allows safe device sharing among guest virtual machines. Their approach also does not require any modifications of applications or kernel drivers.

VMM-bypass I/O, however, heavily depends on the specialized hardware support. Raj et al. suggested a similar approach for high performance network virtualization using a customized network processor or a specialized core in a multi-core system [24]. Willmann et al. modified RiceNIC to support virtualization and implemented multiple transmitting and receiving paths and send/receive queues for multiple guest domains [31]. Some recent 10 Gbps Ethernet adapters also support multiple transmitting and receiving paths, along with multiple send/receive queues that obviate the use of bridges and allow multiple guest domains to use the network device simultaneously [8, 10, 30]. These researches require new hardware devices for better network performance, while our study focuses on software solutions when using traditional Ethernet devices.

Our approach to gbuffer borrows many concepts from previous studies on preallocated shared memory, which are aimed at accelerating IPC and network communication. Govindan et al. found that user/kernel domain switches and mapping switches between different user virtual address spaces show substantial overheads and increase IPC latency [11]. They proposed a new kind of IPC mechanism, memory-mapped streams (MMS), which uses preallocated shared memory between applications, and achieved better performance.

Druschel et al. suggested *fast buffers* for microkernel-based operating systems [7]. They pointed out that moving data across the multiple protected domains needs to be efficient for high performance communication. They used immutability of I/O buffers, which means that I/O buffers may not be changed once they are written, and they improved the network communication performance of the Mach kernel.

Several researchers exploited preallocated shared memory for fast and efficient packet capturing. Deri presented a new kind of socket, `PF_RING`, and *libpcap* library for Linux [6]. They exported kernel-space ring buffers through `mmap` in order to allow user-space applications to access the data in the ring buffers without system calls. Bos et al. utilized preallocated shared memory to make more than two user-space applications efficiently process the same packets in the buffers [5]. When more than two applications try to process the packets, the packet in one virtual address space must be copied to the other virtual address space. Instead of copying packets, they used the preallocated shared memory called *PBuf* and displayed improved packet processing throughput.

While our research focuses on the communication between sender and receiver, which are not on the same machine, some studies have been presented to improve inter-domain communication performance. Kim et al. proposed *XWAY*, which accelerates communication between domains on the same physical machine [16]. They replaced the existing TCP/IP protocol stack with the lightweight communication protocol without modifications in Xen. This leads to the reduced communication path and high network performance. Zhang et al. also used similar methods to achieve better interdomain socket communication [32].

6 Conclusion

The infrastructure and energy cost of operating a datacenter are estimated to be three times greater than the IT hardware cost in year 2014 [4], and emerging multicore technology increases the idle time of processing elements in each server node. Virtualizing the existing distributed systems or parallel processing systems reduces the energy, cooling, maintenance, and hardware purchasing cost. It also improves the availability of the entire systems by providing live-migration of server nodes. Therefore, most of the parallel and distributed systems will be virtualized in the near future.

In the virtualized environment, the peripheral I/O devices, such as network interface cards, are to be shared by multiple virtual machines simultaneously. However, because the current network interface cards are not designed to be concurrently used by multiple virtual machines, the virtual machine monitor must manage access to the network devices from virtual machines and this result in the significant performance overhead. In particular, the parallel and distributed systems have frequent networking operations between the nodes in the systems; this overhead may induce the critical performance degradation by reducing the processor time that could be given to other processor-bound tasks.

Some of the major causes of this networking overhead, such as the cross-domain data copy, exist to provide performance isolation and security protection between virtual machines. Although both the isolation and protection between virtual machines are strong points of the virtualization technology, they are of little value in parallel and distributed systems, which are our targets. Based on this philosophy, we introduced a novel virtual networking architecture, named SETH that reduces the performance overhead of the virtualized network device driver.

The SETH architecture features the front-end device driver and back-end device driver, which employ shared buffer memory to transfer network packets between the normal guest systems and isolated device driver guest system that masters and controls the physical network interface card. The prototype of the suggested approach is implemented in the Xen hypervisor and paravirtualized Linux kernel to be evaluated. In our evaluation, it shows up to 9% bandwidth improvement when transmitting large packets, and up to 30% of processor utilization reduction when it receives small packets continually. Also, it enhances the performance of real-world workloads by up to 12% in the evaluation. With these results, we can tell that our solution reduces significant performance overhead, as well as improves network bandwidth in the virtualized environment.

Ultimately, SETH modifies only the fundamental device driver layer, and it is not seen from the upper layer. Therefore, we can combine the existing research results, which aim to reduce network virtualization, with our approach to obtain synergy effects. We will explore the possibility to reduce the overhead further using SETH alone by applying the existing research results with consideration to the characteristics of the target systems.

Acknowledgements This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (R01-2007-000-11832-0) and also by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2008-314-D00345).

References

1. Amza C, Cecchet E, Chanda A, Cox A, Elnikety S, Gil R, Marguerite J, Rajamani K, Zwaenepoel W (2002) Specification and implementation of dynamic web site benchmarks. In: Proceedings of IEEE 5th annual workshop on workload characterization
2. Apparao P, Makineni S, Newell D (2006) Characterization of network processing overheads in Xen. In: Proceedings of IEEE int'l workshop on virtualization technology in distributed computing, Nov 2006
3. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauery R, Pratt I, Warfield A (2003) Xen and the Art of Virtualization. In: Proceedings of ACM symp. operating systems principles, Oct 2003, pp 164–177
4. Belady C (2007) In the data center, power and cooling costs more than the IT equipment it supports. In: Electronics Cooling, Feb 2007
5. Bos H, Bruijn WD, Cristea M, Nguyen T, Portokalidis G (2004) FFPF: fairly fast packet filters. In: Proceedings of 6th symp. operating systems design and implementation, Dec 2004, pp 24
6. Deri L (2004) Improving passive packet capture: beyond device polling. In: Proceedings of int'l system administration and network engineering conf., Sep 2004
7. Druschel P, Peterson LL (1993) Fbufs: a high-bandwidth cross-domain transfer facility. In: Proceedings of ACM symp. operating systems principles, Dec 1993, pp 189–202
8. Enabling virtualization in the datacenter. White paper, Neterion, Jan 2007
9. Evangelinos C, Hill CN (2008) Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. In: Proceedings of cloud computing and its applications
10. GadelRab S (2007) 10-Gigabit ethernet connectivity for computer servers. IEEE Micro 27(3):94–105
11. Govindan R, Anderson DP (1991) Scheduling and IPC mechanisms for continuous media. Proc ACM SIGOPS Oper Syst Rev 25(5):68–80
12. Govindan S, Nath AR, Das A, Urgaonkar B, Sivasubramaniam A (2007) Xen and Co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In: Proceedings of int'l conf. virtual execution environments, Jun 2007, pp 126–136
13. Henning JL (2006) SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput Archit News 34(4):1–17
14. Kamp P-H, Watson RNM (2000) Jails: confining the omnipotent root. In: Proceedings of int'l system administration and networking conference, May 2000
15. Keahey K, Figueiredo R, Fortes J, Freeman T, Tsugawa M (2008) Science clouds: early experiences in cloud computing for scientific applications. In: Proceedings of cloud computing and its applications
16. Kim K, Kim C, Jung S-I, Shin H-S, Kim J-S (2008) Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In: Proceedings of ACM SIGPLAN/SIGOPS int'l conf. virtual execution environments, Mar 2008, pp 11–20
17. Koomey J (2006) Estimating total power consumption by servers in the U.S. and the world. Technical report, Lawrence Berkeley National Laboratory
18. Kopytov A (2009) SysBench: a system performance benchmark. http://sysbench.sourceforge.net/
19. Liu J, Huang W, Abali B, Panda DK (2006) High performance VMM-bypass I/O in virtual machines. In: Proceedings of 2006 USENIX annual technical conf., May 2006, pp 1–3
20. Menon A, Cox AL, Zwaenepoel W (2006) Optimizing network virtualization in Xen. In: Proceedings of 2006 USENIX annual technical conf., Jun 2006, pp 15–28
21. Menon A, Santos JR, Turner Y, Janakiraman G, Zwaenepoel W (2005) Diagnosing performance overheads in the xen virtual machine environment. In: Proceedings of ACM/USENIX int'l conf. virtual execution environments, Jun 2005, pp 13–23

22. Padala P, Zhu X, Wang Z, Singhal S, Shin KG (2007) Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59, HP
23. Price D, Tucker A (2004) Solaris zones: operating systems support for consolidating commercial workloads. In: Proceedings of 18th large installation system administration conf., Nov. 2004, pp 241–254
24. Raj H, Ganev I, Schwan K, Xenidis J (2006) Self-virtualized I/O: high performance, scalable I/O virtualization in multi-core systems. Technical Report GIT-CERCS-06-02, CERCS, Georgia Tech
25. Santos JR Janakiraman G, Turner Y (2007) Xen network I/O: Performance Analysis and Opportunities for Improvement. Xen Summit Spring 2007. http://xen.xensource.com/files/xensummit_4/NetworkIO_Santos.pdf, Apr
26. Singh R (2007) Server virtualization and consolidation — a case study. White paper. http://www.ibm.com/support/techdocs, IBM
27. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L (2007) Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of ACM SIGOPS/Eurosys European conf. on computer systems, Mar. 2007, pp 275–287
28. Sugerman J, Venkitachalam G, Lim B-H (2001) Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In: Proceedings of 2001 USENIX annual technical conf., Jun 2001, pp 1–14
29. Waldspurger CA (2002) Memory resource management in VMware ESX server. In: Proceedings of symp. operating systems design and implementation, Dec 2002, pp 181–194
30. Virtual Machine Device Queues. White paper, Intel, 2007
31. Willmann P, Shafer J, Carr D, Menon A, Rixner S, Cox AL, Zwaenepoel W (2007) Concurrent direct network access for virtual machine monitors. In: Proceedings of IEEE int'l symp. high performance computer architecture, Feb 2007, pp. 306–317
32. Zhang X, McIntosh S, Rohatgi P, Griffin JL (2007) XenSocket: a high-throughput interdomain transport for virtual machines. In: Proceedings of ACM/IFIP/USENIX int'l middleware conf., Aug 2007, pp 184–203