Analysis of Network I/O Performance in KVM

RYOTA OZAKI† and AKIHIRO NAKAO ††,†

Recent virtualization technologies combine para-virtualization and hardware-assisted virtualization techniques as so-called hybrid-virtualization. Para-virtualization and hardware-assisted virtualization have performance issues in different aspects, respectively, but combined use of both sometimes compensates for the issues of each, thus, potentially gains better performance. Although KVM acquires reasonable I/O performance through this hybrid-virtualization approach, we believe its I/O performance has room for improvement. To this end, we investigate how KVM handles network I/O, report its recent improvement with a para-virtualized network driver, and finally describe a perspective toward further improvement based on the investigation.

1. Introduction

x86 virtualization techniques have recently evolved in two ways—*para-virtualization* and *hardware-assisted virtualization*—to mitigate the performance overhead incurred by virtualizing physical resources. For example, Xen achieves its high performance by para-virtualization—a technique to modify guest operating systems (OS) to suit their hypervisor, while several processor vendors enhanced their processors to assist virtualization. This processor extension facilitates full-virtualization—virtualization that allows a guest OS to run unmodified—with reasonable performance.

Both of these approaches, however, still leave room for improvement in performance, thus, a hybrid approach combining the above two virtualization techniques, so-called *hybrid-virtualization*[5], is proposed to gain more performance.

KVM[3] is one of the virtual machine monitor (VMM)—a thin software layer to enable virtualization—implementations that adopt the hybrid-virtualization. KVM is designed as a host-based VMM configuration that virtual machines (VMs) are deployed on an ordinary OS (**Fig. 1**).

One of the most important pros of a host-based VMM technique is that it utilizes the host OS's capabilities, e.g., scheduling, memory management and power management, so it can leverage the most recent developments of the host OS that include various hardware supports, e.g., for new architectures and for new devices, as well as software developments, e.g., for resource allocation/management and scheduling. For example, it could achieve good scalability in terms of the number of guest OSes that may run with reasonable performance if guests' memories are swappable through the host OS's virtual memory management.

KVM has been originally designed to provide full-virtualization using hardware assistance. But, unfortunately, performance of the guest OSes in KVM's initial implementation turns out not great, especially under I/O intensive operations. Although, after the recent development, especially with the introduction of hybrid-virtualization, KVM achieves reasonable I/O performance, we yet believe its I/O performance still leaves room for further improvement, thus, set our goal to identify the bottlenecks in its network I/O and investigate how we can resolve them.

To this end, this paper reports our recent investigations into KVM's network I/O performance, both in full-virtualized configuration and in para-virtualized one, and our attempts to reveal how KVM handles network I/O. Furthermore, we describe a perspective toward further improvement based on the investigations.

The rest of this paper is organized as follows. Section 2 describes virtualization technologies in x86 architectures. Section 3 and Section 4 describe an overview of KVM and its para-virtualization, respectively. Section 5 and Section 6 report evaluation and analysis of network I/O processing in KVM, respectively. Section 7 discusses further improvement of KVM and

† New Generation Network Research Center, National Institute of Information and Communications Technology
†† Interfaculty Initiative in Information Studies, Graduate School of Interdisciplinary Information Studies, The University of Tokyo

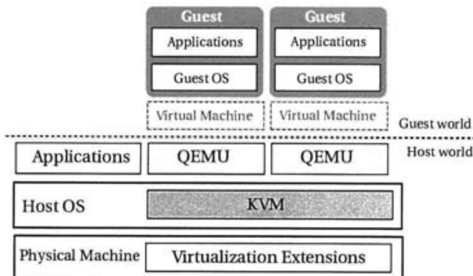


Fig. 1 KVM configuration.

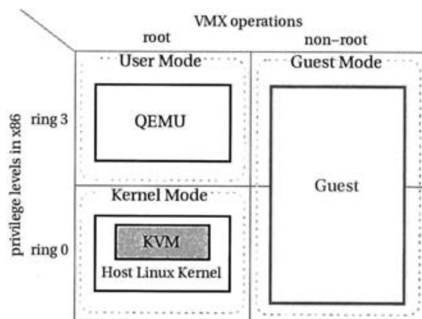


Fig. 2 Execution modes in KVM.

Section 8 concludes the paper.

2. Recent x86 Virtualization Evolutions

Since early generations of x86 processors lacked hardware support to facilitate virtualization, former VMMs used to compensate for the lack through software techniques, such as dynamic translation[6] and para-virtualization[8]. Nowadays, most x86 processors have enabled hardware support for virtualization[1),7] so that VM developers can implement VMs in a simpler and more efficient manner than ever.

2.1 Para-virtualization

Para-virtualization aims at minimizing overhead in virtualization through cooperation between guest OSes and a VMM. Para-virtualization often enables a guest OS to perform comparable to an OS running on a bare machine, e.g., Xen[8], one of the VMM implementations with para-virtualization, performs as well as a native Linux in some cases.

In order to achieve the cooperation between guest OSes and a VMM, guest OSes are often modified to run on the VMM, for example, privileged CPU instructions, memory and time operations, etc. are modified in the guest OSes. *Hypercall* APIs are provided for the modified guest OSes to use VMM functions and to communicate with the other guest OSes. The hypercalls are similar to system calls for applications in an ordinary OS.

One of the cooperations between a guest OS and a VMM is reduction of redundant processing in the VMM and the guest OS. For example, the VMM eliminates timer-tick interrupts while the guest OS is idling. For another example, the guest OS can assemble multiple hardware accesses into one hypercall.

A caveat, however, is that para-virtualization sometimes introduces unwanted overhead, such as a redundant path in system calls. In Xen, a guest kernel run in *ring 1* in the x86 architecture, so that system calls issued by guest applications are first handled in Xen running in *ring 0*. As a result, Xen cannot implement system calls via *fast-path*, i.e., the system calls cause transitions only between the guest applications and the guest kernel.

As an aside, Xen is originally designed to provide para-virtualized VMs, but currently it also provides full-virtualized VMs using PC emulator and hardware assistance described in the next section. Furthermore, Xen now introduces hybrid-virtualization technique[5] as explained in the following section.

2.2 Hardware-assisted virtualization

Several x86 processor vendors have recently enhanced their processors so that VM developers may easily fully-virtualize guest OSes with reasonable performance. Intel VT-x[7] and AMD-V[1] are the most popular implementations of such a hardware assistance for x86 virtualization. Since they are similar in their capabilities to each other, in this paper, we discuss Intel VT-x as a representative of the x86 hardware-assisted virtualization technologies.

Intel VT-x provides the following capabilities:

- A new operation for guest OSes
  VT-x provides two additional operations, VMX *root* and *non-root* operations, orthogonal to traditional privilege levels in x86 architectures. VMMs, such as a host OS or a hypervisor, run in the root operation and guest OSes run in the non-root operation.
- Hardware switch between the VMX operations
  A state transition from a root operation mode to a non-root one is called a *VM entry*, and the reverse is called a *VM exit*. While a VM entry is explicitly called by

a VMM, a VM exit is triggered by special events, e.g., privileged instructions, interrupts and exceptions, in the non-root operation mode.

- Exit reason reporting
  After the VM exit, CPU returns back to the VMM. The VMM may examine the exit reason from CPU and perform some appropriate actions along the reason.

Since the virtualization extension only provides CPU virtualization, VMMs still need to virtualize MMU and peripheral devices. Several implementations of full-virtualization with hardware assistance utilizes a PC emulator, e.g., QEMU[2].

2.3 Hybrid-virtualization

Hybrid-virtualization employs the above two techniques, para-virtualization and hardware-assisted full-virtualization. This hybrid approach yields both high-performance and simplicity of implementation with a few modifications to guest OSes, since each of the two techniques may compensate for some of the disadvantages of each other.

For instance, difficulty in implementing fast system-calls in para-virtualization would be mitigated with hardware assistance, since with the presence of hardware assistance, system calls could be issued by guest OSes through fast paths without VMM's intervention. For another example, support for a variety of peripheral devices in hardware-assisted full-virtualization would be unnecessary in combination with para-virtualization, since guest OSes might only need to para-virtualize the devices in question.

VMM technologies that have recently emerged with this hybrid virtualization technique include Xen (recent versions) and KVM[5].

3. An Overview of KVM

KVM has originally been designed to provide full-virtualized VMs combining with a PC emulator and with hardware assistance. KVM itself only virtualizes guest memories and IRQs, while the other peripheral components to be virtualized are taken care of by a PC emulator.

3.1 KVM components

KVM is a loadable kernel module extension to a Linux kernel. It manages VM instances, virtual CPUs and virtual IRQs. It provides a character device driver interface, `/dev/kvm`, to the user space, so that user processes may control VMs, virtual CPUs and IRQs in KVM using `ioctl` system call or a `libkvm` library.

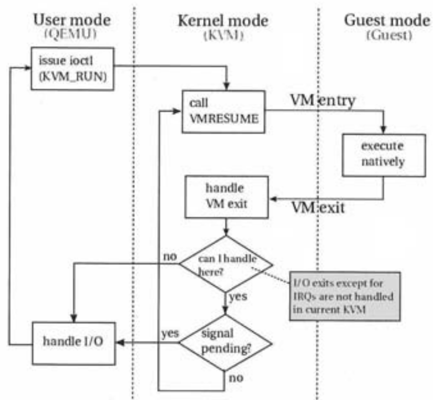


Fig. 3 A guest execution loop.

KVM provides the following APIs:

- create/destroy VMs,
- `mmap` virtual memories of VMs,
- read/write registers of virtual CPUs and IRQs (IOAPIC, Local APIC and PIC), and
- inject interrupts to virtual CPUs.

Current KVM utilizes QEMU[2] as a PC emulator in the user space (Fig. 1). QEMU provides most emulations required for full-virtualization, for example, PCI bus and peripheral device emulations, their I/O handling, and interrupt injections.

3.2 Execution modes

KVM provides a new execution mode, *guest mode*, in addition to *user mode* and *kernel mode* in ordinary OSes. KVM, QEMU and guest OSes run in the kernel mode, the user mode, and the guest mode, respectively (Fig. 2). The guest mode altogether is equivalent to the non-root operation in Intel VT-x, namely, guest kernels run in ring 0 *and* non-root operation, and guest user processes runs in ring 3 *and* non-root operation.

Figure 3 shows the execution path of a guest kernel. The path indicates that a mode transition between a guest and QEMU needs to pass through KVM so that virtualization processing in QEMU always incurs more overhead than that in KVM.

3.3 Guest memory management

KVM allows a guest to have its own individual pseudo physical memory. A guest OS can manage its memory in a traditional manner, while KVM actually intercepts guest's operations for memory management such as instruc-

tions on CR3 register and on page tables, etc. KVM manages a shadow copy of the guest's page table, and the shadow copy actually has an effect on MMU.

KVM provides the `mmap` system call to expose guest physical memory to QEMU. QEMU maps the entire guest physical memory to its own virtual memory space when the guest boots up☆. QEMU may conduct DMA emulations and optimizations of guest memory accesses, e.g., para-virtualized device drivers may utilize this interface.

3.4 Network processing

KVM implements fully-virtualized network drivers via QEMU. QEMU provides emulated network interface cards (NICs) and a virtual LAN to serve network I/O requests from guest OSes.

An I/O request in a guest OS causes a VM exit and the exit is initially handled by KVM. KVM then passes a request to QEMU with a callback function☆☆ and QEMU processes the request.

DMA data transfer is performed via the mapped memory between QEMU and a guest OS described in Section 3.3. The address of the DMA memory area of a guest OS device will be handed to QEMU via VM exit at its boot time.

When QEMU handles an I/O request for the transfer from/to the DMA memory area, QEMU copies data to/from its own buffer and then injects an interrupt for the DMA completion into the guest using KVM's API.

4. Para-Virtualization in KVM

KVM only para-virtualizes a block device and a network device. Para-virtualization in KVM requires only the installation of driver modules so that guest kernels may not need to be recompiled.

4.1 Para-virtualized network drivers

KVM has several versions of prototype implementations of para-virtualized (PV) network drivers, because the implementation of the PV drivers has been under work-in-progress and they are not officially included in the KVM main development tree yet. Although each PV driver is implemented with a different goal, the

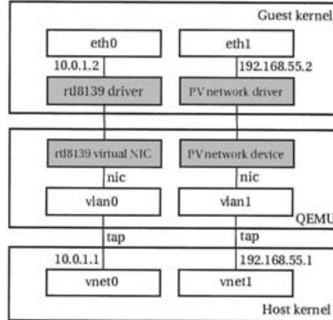


Fig. 4 Network driver configuration and VLAN configuration in QEMU.

architectures of these PV drivers are essentially the same.

A PV network driver has a corresponding PV network device in QEMU (**Fig. 4**). The PV device uses mapped memory for exchanging data with the PV driver. Since the PV network driver and the PV device cooperate with each other, they can eliminate unnecessary I/O port accesses and switches between a guest OS and a VMM in transferring data in bulk.

4.2 Hypercalls

KVM has a capability of hypercalls described in Section 2.1 as well as Xen. KVM has two types of hypercalls; one is a *user hypercall* handled in QEMU, and the other is a *kernel hypercall* handled in KVM.

An early implementation of the para-virtualized network driver, which is in `kvm-36`, uses a user hypercall for a notification from a PV network driver to a PV network device, while a recent implementation of the para-virtualized network driver, which is in `kvm-51`, uses an I/O port access for the notification instead of a hypercall.

5. Evaluation

5.1 Experiment setup

We conduct several experiments to evaluate KVM network I/O performance with the two versions of KVM, `kvm-36` and `kvm-51` that implement prototype PV drivers☆☆☆.

Table 1 shows the host configuration used in the experiment. We use `iperf-2.0.2-3.fc7` and `ping` commands in `iputils-20070202-3.fc7`

☆ KVM limits a guest physical memory up to 2.047 Gbytes on 32-bit hosts to enable QEMU to map the entire guest physical memory into its virtual memory space.

☆☆ The callback function is implemented as just a return from `ioctl`.

☆☆☆ Note that the PV driver that is used with kvm-51 is not actually implemented for kvm-51. We have imported a patch set of the PV driver to kvm-51, because later versions of KVM do not work in our environment due to a bug.

Table 1 Experiment Setup

CPU	
Model	Intel Pentium D 930 (SL95X)
# of cores	2
Clock	3.0 GHz
FSB	800 MHz
L1 data cache	16 KB
L2	16 KB (per core)
Extensions	Virtualization, EM64T, SpeedStep

Main memory	
Capacity	4,096 MB

OS	
Host	Fedora 7 (linux-2.6.23-rc3)
Guest (kvm-36)	Fedora 7 (linux-2.6.23-rc3)
Guest (kvm-51)	Fedora 7 (linux-2.6.24-rc3)

Table 2 Cost of Hypercalls

Kernel hypercall	3.93 μsec.
User hypercall	6.76 μsec.

in our evaluation. We deploy iperf and ping both in the guest OS and in the host OS. The guest OS and the host OS communicates with each other over the VLAN in QEMU as illustrated in **Figure 4**. Note, throughout these evaluations, the SpeedStep feature☆ has been turned off.

5.2 Cost of hypercalls

This section evaluates the cost of two hypercalls in KVM. We implement a null(empty) hypercall handler both in QEMU and in KVM, respectively, and deploy a benchmark program in a guest kernel as a kernel module. The module issues hypercalls to QEMU and KVM from the guest kernel. A kernel hypercall runs through (i) guest kernel (ii) KVM (iii) guest kernel, while a user hypercall runs through (i) guest kernel (ii) KVM (iii) QEMU (iv) KVM (v) guest kernel.

Table 2 shows the cost of the hypercalls is attributed to the round trip time (RTT) between the guest kernel and KVM/QEMU. Some functions of KVM are implemented in a host kernel (KVM) while others are in a user process (QEMU). Those implemented in QEMU will suffer from the extra overhead represented by the difference in RTTs shown above, compared to those implemented in KVM.

5.3 TCP bandwidth

This section evaluates TCP bandwidth with various TCP window sizes. We perform two iperf runs: sending packets from an iperf client on a host OS to an iperf server on a guest OS (RX), and vice versa (TX). We evaluate four drivers; RealTek RTL-8139 ethernet driver (rtl8139) and a para-virtualized network driver (pv) in kvm-36, and those in kvm-51.

☆ The SpeedStep is a function to change its clock speed dynamically. In Linux, the speed is changed by a daemon process that periodically checks CPU loads.

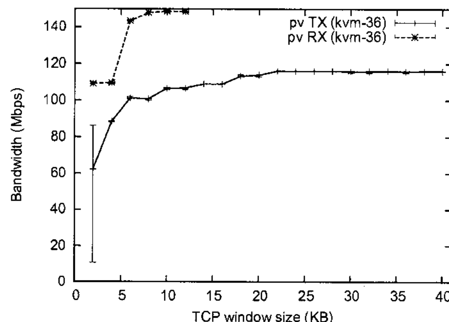


Fig. 5 TCP window size v.s. bandwidth (pv in kvm-36).

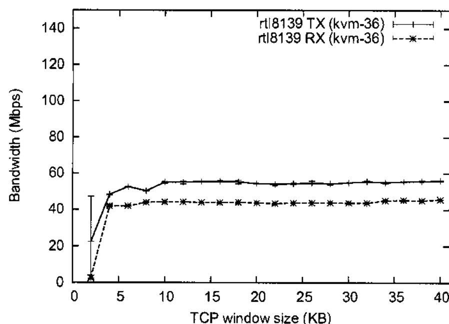


Fig. 6 TCP window size v.s. bandwidth (rtl8139 in kvm-36).

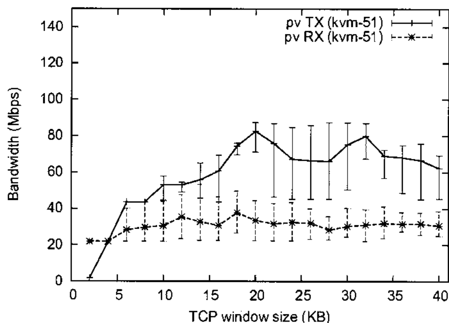


Fig. 7 TCP window size v.s. bandwidth (pv in kvm-51).

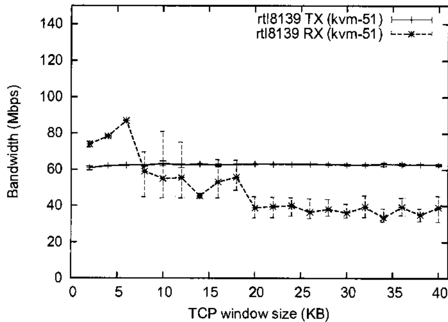


Fig. 8 TCP window size v.s. bandwidth (rtl8139 in kvm-51).

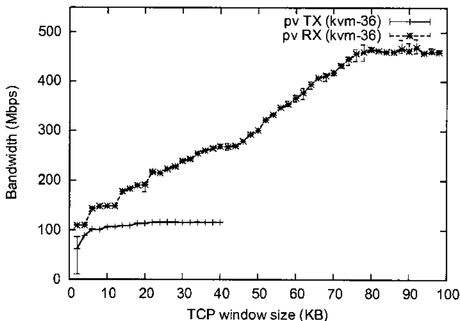


Fig. 9 TCP window size v.s. bandwidth (pv in kvm-36).

Note that a baseline bandwidth is 1.6 Gbps, which is obtained from the iperf measurement where client and server run on the host OS.

Figure 5, 6, 7, 8, 9 show the results of this experiment.

We increase the TCP window size until the bandwidth saturates. As shown in **Figure 6**, RX bandwidth of pv in kvm-36 constantly increases until the TCP window size reaches 80 KB, while, as in **Figure 7**, the other bandwidth curves saturated at less than 40 KB. As shown in **Figure 9**, the maximum RX bandwidth of the pv achieved in kvm-36 is 469 Mbps. However, **Figure 7** shows that bandwidth of the pv in kvm-51 reaches only up to 82.5 Mbps. The reason of this result could be that the pv in kvm-51 is implemented fairly recently and has not been optimized for performance yet. Developers of KVM, however, have reported that the performance of pv has greatly been improved with recent modifications that have been added. We intend to conduct further evaluation using

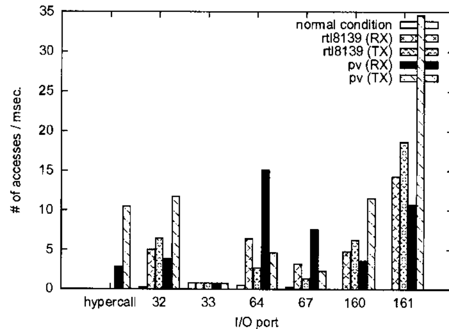


Fig. 10 The number of I/O port accesses and hypercalls.

this improved implementation.

6. Analysis Network I/O processing in KVM

6.1 A case of a full-virtualized driver

We pick up a network driver for rtl8139 ethernet card for the evaluation of a full-virtualized driver. The rtl8139 driver has the following features:

- maximum size of ethernet frames: 1,536 bytes
- number of concurrent DMA data transmission: 4
- receive buffer size: 32 Kbytes

As shown in **Figure 10**, packet transmission and reception cause many I/O accesses. All the accesses cause VM exits, each of which invokes two transitions between the non-root operation and the root one and vice versa.

Figure 11 (TX) shows that I/O port accesses in processing a packet transmission as follows. First, a socket buffer from a network stack is copied to a DMA buffer, and then the DMA controller in rtl8139 gets triggered. When the transmission is completed, the controller issues an interrupt. If only one packet is transmitted through NIC, the total number of I/O accesses is three.

Processing a packet reception is more complex than the transmission. The rtl8139 driver implements the NAPI framework☆. The NAPI framework enables network drivers to process packet reception more efficiently than ever by eliminating redundant interrupts and expensive interrupt handling, and disabling them

☆ The NAPI framework is a new design concept of network drivers in Linux kernel.

(TX)

(1) *set data to a DMA buffer*
(2) W: fire DMA
(3) *interrupt* (tx completed)
(4) R: interrupt status
(5) W: ACK (to all tx completions)

(RX)

(1) *interrupt* (packet arrived)
(2) R: interrupt status
(3) W: disable interrupts
(4) *rx poll* (called from software interrupt)
(5) R: rx interrupt?
(6) while (R: not empty?)
  (a) receive packet
  (b) if (R: rx interrupt?) then W: ACK
(7) W: enable interrupts

'R:' and 'W:' indicate read/write operations from/to a I/O port respectively.

Fig. 11 I/O accesses in full-virtualized driver processing.

while handling incoming packet. Thus, network drivers can handle multiple incoming packets with one interrupt.

Figure 11 (RX) shows I/O port accesses in processing a packet reception.

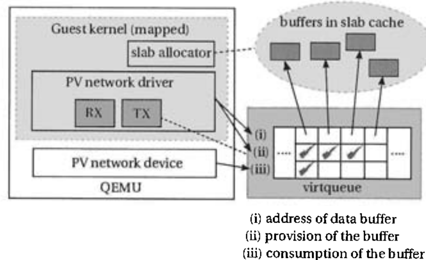
The rate of I/O port accesses (the number of accesses per packets) in a packet reception is greater than that in a packet transmission. If packets come in apart in time, the total number of I/O accesses is up to seven. On the other hand, if incoming packets are bursty, the total number of I/O accesses could be close to three.

Without virtualization, NAPI might achieve efficient processing by eliminating expensive interrupt handling, but in a virtualized environment, NAPI cannot eliminate I/O port accesses so that packet reception processing may end up being unable to achieve sufficient performance.

6.2 A case of a para-virtualized driver

PV network driver and device are implemented using *VirtIO*, a common architecture and API for virtual I/O implementations in para-virtualization.

VirtIO consists of tree conditions; the first is that a guest and a VMM can cooperate with each other, the second is that they share some memory areas, and the third is that they provide a mechanism to notify to the other side. It uses *virtqueue* that is the shared page between them in order to notify (i) addresses of data buffers (ii) provisions and (iii) consumptions of the buffers (**Fig. 12**).



(i) address of data buffer
(ii) provision of the buffer
(iii) consumption of the buffer

Fig. 12 Para-virtualized driver and device.

Data exchanges are performed not with the virtqueue but through another memory area. In KVM, the PV driver and device share memory as described in Section 3.3 so that the PV network device can access memory areas not only of the PV network driver but also of the entire guest kernel. `sk_buff`☆ is used as data buffers to exchange between the PV driver and device, in fact, buffers in the `sk_buff` that are allocated in the guest kernel in advance are used for the data transfer from the PV network device to the PV driver and vice versa.

When a virtqueue in one side gets filled or a pre-set timer expires, one side notifies the other side to let it consume the buffer using a specific I/O port access or a hypercall in a transmission, and an interrupt in a reception.

6.3 Pros and Cons of a para-virtualized network driver

A PV network driver have two advantages over drivers for physical NICs: First, a PV network device can be equipped with the ideal NIC capabilities such as pseudo scatter/gather DMA. The pseudo scatter/gather DMA is known to gain better performance than the normal pseudo DMA[4]. It could also enable bulk data transfer of larger size than the on-chip buffer and MTU on the physical NICs.

Second, the PV driver can avoid redundant I/O port accesses. The PV driver and device will share memory areas so that they can read/write data and each status without any I/O requests. This eliminates VMM interferences that incur operation transitions and mode switches.

Unfortunately, these advantages are effective only within a virtualized environment. If an application sends data to outside the physical host, the efficiency of the data transfer is lim-

☆ `sk_buff` is data structure in Linux networking. The `sk_buff` is able to contain fragmented data buffers.

Table 3 TCP bandwidth (Mbps)

with in-kernel IRQ handling				
	ave.	min	max	σ
RX	74.3	73.0	75.5	0.626
TX	44.9	44.8	45.1	0.119
without in-kernel IRQ handling				
	ave.	min	max	σ
RX	69.6	67.8	70.6	0.772
TX	39.9	39.8	40.1	0.110

ited by the capabilities of the physical NICs.

7. Consideration towards further improvement

Our experiment results have confirmed that network I/O performance in KVM has greatly improved using PV drivers, however, we believe that the performance still leaves room for further improvement.

For example, KVM in the host kernel should implement PV devices within itself, not in the user space. A current PV device implementation requires redundant two mode switches and a packet copy between KVM and QEMU in both send and receive paths. We strongly believe that in-kernel PV devices could avoid this inefficient path, and could improve the network I/O performance further.

For the feasibility study for this conjecture, we conduct the following experiment. **Table 3** shows a bandwidth comparison between two configurations in KVM. One configuration enables in-kernel IRQ handling, while the other configuration disables it. **Table 3** shows that the former bandwidth is superior than the latter, by 6.5% in RX and by 12.5% in TX.

A difference between processing in these two configurations is where I/O exits caused by the accesses to PIC get handled. In the former case, the I/O exits are handled in KVM so that only transitions between a guest kernel and a host kernel may occur. On the other hand, in the latter case, I/O exits are handled in QEMU so that transitions between the host kernel and QEMU may occur in addition to the above transitions.

This indicates the in-kernel PV driver has a potential to improve network performance more or less by 10%. In addition, the in-kernel PV driver is expected to eliminate data copy between QEMU and the host kernel.

8. Conclusion

KVM is still not a mature VMM implementation yet, compared with the other VMM implementations. Nonetheless, its potential has recently attracted lots of attentions in the community. Especially, hybrid-virtualization in KVM is expected to overcome inefficient I/O performance of guest OSes.

This paper has made two contributions: one that we have unveiled network processing of KVM both on full-virtualization and on para-virtualization, the other that we have evaluated and analyzed KVM's network I/O performance in these two cases and also propose possible further improvement based on the results.

We intend to continue investigating how KVM handles network I/O and its performance in more detail. We also plan to implement a proposed method described in Section 7 and to evaluate its effect.

References

1) AMD: AMD64 Virtualization Codenamed "Pacifica" Technology, Secure Virtual Machine Architecture Reference (2005).
2) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, Proc. *USENIX 2005 Annual Technical Conference, FREENIX Track*, pp.41–46 (2005).
3) Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, Proc. *Ottawa Linux Symposium 2007 (OLS '07)*, pp.225–230 (2007).
4) Menon, A., Cox, A.L. and Zwaenepoel, W.: Optimizing Network Virtualization in Xen, Proc. *2006 USENIX Annual Technical Conference (USENIX '06)*, pp.15–28 (2006).
5) Nakajima, J. and Mallick, A.K.: Hybrid-Virtualization — Enhanced Virtualization for Linux, Proc. *Ottawa Linux Symposium 2007 (OLS '07)*, pp.87–96 (2007).
6) Sugerman, J., Venkitachalam, G. and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, Proc. *2001 USENIX Annual Technical Conference (USENIX '01)*, pp.1–14 (2001).
7) Uhlig, R., Neiger, G., Rodgers, D., Santoni, A.L., Martins, F.C.M., Anderson, A.V., Bennett, S.M., Kagi, A., Leung, F.H. and Smith, L.: Intel Virtualization Technology, IEEE Computer, Vol.38, No.5, pp.48–56 (2005).
8) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, Proc. *the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp.164–177 (2003).