When I/O Interrupt Becomes System Bottleneck: Efficiency and Scalability Enhancement for SR-IOV Network Virtualization

Jian Li, Shuai Xue, Wang Zhang, Ruhui Ma, Zhengwei Qi, and Haibing Guan, *Member, IEEE*

Abstract—High performance networking interface cards (NIC) have become essential networking devices in commercial cloud computing environments. Therefore, efficient and scalable I/O virtualization is one of the primary challenges on virtualized cloud computing platforms. Single Root I/O Virtualization (SR-IOV) is a network interface technology that eliminates the overhead of redundant data copies and the virtual network switches through direct I/O in order to achieve nearly natural I/O performance. However, the SR-IOV still suffers from serious problems due to the high overhead for processing excessive network interrupts as well as the unpredictable and bursty traffic load in high-speed networking connections. In this paper, the defects of SR-IOV with 10 Gigabit Ethernet networking are studied first and two major challenges are identified: excessive interrupt rate and single threaded virtual network driver. Second, two interrupt rate control optimization schemes, called coarse-grained interrupt rate (CGR) control and adaptive interrupt rate (AIR) control are proposed. The proposed control schemes can significantly reduce the overhead and enhance the SR-IOV performance compared with the traditional driver with fixed interrupt throttle rate (FIR). In addition, multi-threaded VF driver (MTVD) is proposed that allows the SR-IOV VFs to leverage multi-core resources in order to achieve high scalability. Finally, these optimizations are implemented and detailed performance evaluations are conducted. The results show that CGR and AIR can improve the throughput by 2.26x and 2.97x while saving the CPU resources by 1.23 core and 1.44 core, respectively. The MTVD can achieve 2.03x performance with additional 1.46 cores consumption for VM using the SR-IOV driver.

Index Terms—Virtualization, virtual machine, network virtualization, SR-IOV, 10 Gigabit ethernet (GE), multi-core

1 Introduction

Virtualization is a key enabling technology for cloud computing infrastructure due to its significant advantages of flexible and reliable resource management. Virtualization allows the platform to maximize the utilization of underlying hardware resources with server consolidation attributes as well as resilient resource allocation and live migration. The network I/O virtualization is one of the most critical techniques in cloud computing, since the common applications running on a cloud computing system are large-scale concurrent network intensive applications such as web servers and storage services [1], [2]. For these applications, the cloud platform is required to provide high throughput and low latency network connection. Consequently, I/O virtualization has become one of the most important components of virtualization infrastructure, whose efficiency severely affects the entire system performance in various situations [3], [4].

Generally, the I/O virtualization is a technology for sharing I/O devices. It can be implemented using either software emulation or hardware assistance. Both solutions are supported in main stream virtual machine monitors (VMM) such as VMware, Xen [5] and KVM [6]. Hardware-based I/O virtualization approaches can significantly improve the performance by allowing the virtual machines(VMs) to directly access the I/O devices. However, these approaches require special hardware supports in order to eliminate the high overhead caused by the redundant data copies and the virtual network switch achieving a high I/O performance [7], [8].

SR-IOV is a hardware-based I/O virtualization approach that defines the shareable I/O devices and offloads the overhead of data copies and the virtual network switch in software-based I/O virtualization model to SR-IOV capable I/O device [9], [10]. Concretely, an SR-IOV capable I/O device can be configured to present multiple Virtual Functions (VFs), and then the device can be natively shared among multiple VMs by assigning different VFs to different VMs. By taking advantages of direct I/O technologies such as I/O memory management unit (IOMMU), each VM can access the device directly without the intervention of VMM. Previous research studies have shown that the SR-IOV can achieve nearly native I/O performance with the hardware I/O assistance [11], [12], [13].

It should be noted that once the SR-IOV has almost eliminated all data transmission overheads using the hardware assisted IOMMU feature, then the virtual interrupt processing

- *J. Li, S. Xue, W. Zhang, and Z. Qi are with School of Software, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {li-jian, mattxue_sjtu, miwa1997, qizhwei}@sjtu.edu.cn.*
- *R. Ma and H. Guan are with the Shanghai Key Laboratory of Scalable Computing and System, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {ruhuima, hbguan}@sjtu.edu.cn.*

Manuscript received 6 July 2016; revised 9 May 2017; accepted 30 May 2017. Date of publication 6 June 2017; date of current version 29 Nov. 2019. (Corresponding author: Jian Li.) Recommended for acceptance by P. Balaji. Digital Object Identifier no. 10.1109/TCC.2017.2712686

overhead becomes the key issue in the virtualization environment, because it incurs multiple costly context switching between the host and the guest causing high CPU overhead and cache pollution. Therefore, the research in this paper focuses on minimization of costly interrupt handling in the virtualization environment. To this end, interrupt coalescing approach is used to reduce the number of interrupts [14], [15]. Gordon et al. proposed a software approach called Exit-Less Interrupts (ELI) method to the host interception in order to reduce the interrupt handling cost [13], [16]. These existing interrupt moderation methods can enhance the virtualized networking performance to be almost as same as bare-metal (∼5.6 Gbps), but still cannot achieve the full line-rate (10 Gbps). Moreover, these methods are unable to adaptively tune the moderation in accordance to the practical network conditions with different applications.

In this paper, the efficient performance and scalability issues in virtualized environment are studied for 10 Gbps SR-IOV Ethernet in multi-core infrastructure. First, experimental analysis performed to demonstrate that the SR-IOV still suffers a serious scalability problem from heavy overhead on processing the excessive interrupts. The indicated problems will be more severe for future network virtualization environments with higher speed Ethernet, such as 40 and 100 Gbps devices. Second, a coarse-grained interrupt rate (CGR) control method and an adaptive interrupt rate (AIR) control method are proposed to adequately adjust the interrupt coalescing rate. The proposed methods can achieve higher performance and superior efficiency of CPU resource utilization compared to the traditional fixed interrupt rate (FIR) control method.

Moreover, it is identified that the SR-IOV performance is also limited by the processing capacity of the single-threaded VF driver for achieving physical line-rate. There are existing network optimizations that employ parallel hardware architecture such as Receive Side Scaling (RSS) and Threaded NAPI [17] (TNAPI) [18]. The RSS distributes the incoming traffic into multiple RX-queues, and the TNAPI allows to create a thread to deal with a virtual Ethernet card per RX queue. However, these optimizations cannot be applied on SR-IOV VFs used in virtual machines. In order to resolve this issue, a multi-threaded VF driver (MTVD) is proposed to parallelize the SR-IOV VF driver so that the network workloads of one VM can be efficiently distributed further across multiple processors to ultimately improve the network performance.

The rest of the paper is organized as follows: In Section 2, the background of SR-IOV and interrupt processing techniques is presented; Section 3 elaborates the investigation on SR-IOV performance and scalability followed by the proposed optimizations for SR-IOV in Section 4; The performance evaluation results are provided in Section 5; Related work is discussed in Section 6. Lastly, the paper is concluded in Section 7.

2 SR-IOV Architecture

2.1 SR-IOV Overview

The SR-IOV is a standard released by the PCI-SIG organization that adopts direct I/O technologies to bypass the VMM during the data movement and uses the IOMMU to offload the memory protection and the address translation to hardware device.

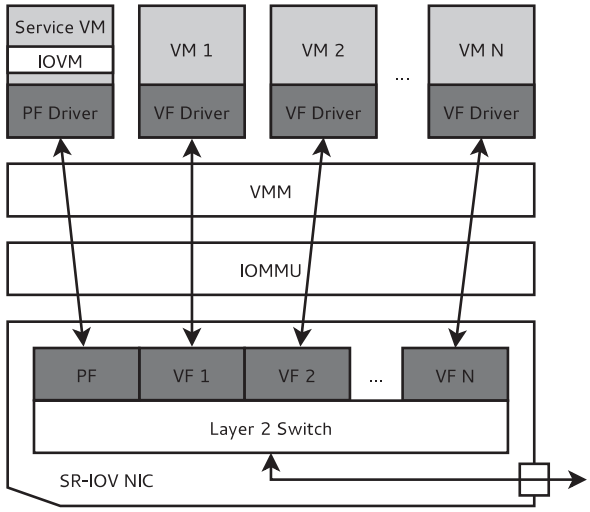


Fig. 1. SR-IOV architecture.

An SR-IOV PCI Express (PCIe) device has one or more Physical Functions (PF) that are standard PCIe functions. Each PF can have multiple Virtual Functions that are "light-weight" PCIe functions. Each VF owns performance-critical resources, such as transmit and receive descriptors, while sharing other major device resources. Thus, each VF can independently receive and transmit packets. Each VF is assigned to a VM, so the VM can access the VF directly and can transmit or receive packets without intervention of the VMM. Fig. 1 demonstrates the architecture of a SR-IOV.

The software components of a SR-IOV NIC driver consist of the PF driver, the VF driver and the SR-IOV manager (IOVM). The PF driver runs in the host OS (or domain 0 in Xen) and has the privilege of access to all device resources. The PF driver is responsible for creating, configuring and managing the VFs. The VF driver runs in a guest OS as a normal network driver and can only access its dedicated VF resources. The VF driver is responsible for performing data movement directly between the guest OS and the VF. The IOVM runs in the host OS along with the PF driver. The host OS cannot enumerate VFs as ordinary devices because they are not full PCIe functions. The IOVM provides a virtual configuration space for the VFs so that the host OS can enumerate and configure them correctly. Then, the VFs can be assigned to guests by the PCI pass-through. Once assigned, the guest OSes can initialize and configure the VFs as ordinary PCIe functions.

A typical SR-IOV Ethernet controller consists of a layer 2 switch, a PF and several VFs. The incoming packets are classified by the layer 2 switch based on MAC addresses and Virtual LAN (VLAN) tags. The packets are then directly transferred into the corresponding VM's receive buffer through DMA. The IOMMU remaps the receive buffer address in the guest OS to a physical address [19]. The VF generates interrupts to notify the VM once the packets are transferred. Each interrupt generated by the VF is captured by the VMM. Then, the VMM injects a virtual interrupt to the corresponding VM for notification of packet arrival. The VF driver in the guest OS is executed to process the packets in the receive buffer. A single interrupt may be raised for multiple incoming packets.

The SR-IOV utilizes the direct I/O technologies in order to achieve near native network performance [12], with lower latency, lighter overhead and higher performance

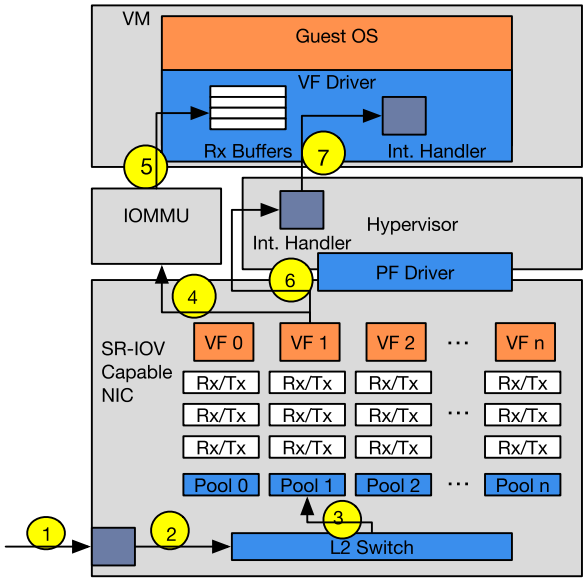


Fig. 2. SR-IOV packet receiving procedure and architecture.

than the traditional I/O virtualization models. Therefore, the SR-IOV has become the default feature in high speed network interface card (NIC), such as >10 Gigabit Ethernet (GE) controller. Moreover, the SR-IOV capable NIC has become the de-facto networking device in cloud infrastructure and datacenter network environments.

2.2 Packet Processing in SR-IOV

In this paper, the primarily concern focuses on the receiver side performance and the scalability of the SR-IOV. This is because the receiving packets are relatively more expensive than the transmitting packets. Therefore, the majority of the performance bottleneck in high speed network environment comes from the receiver side. Normally, the server-side can use up all CPU resources and can suffer the system I/O performance bottleneck, while the packet transmission from clients on another server remain significant CPU vacancies as studied and evaluated in [2], [20].

Here the complexity of packet receiving procedure is explained in detail. Fig. 2 illustrates the receiver side architecture and the packet processing procedure of SR-IOV.

Step 1: Packet propagates through the physical link and arrives at the NIC.

Step 2: Arrived packet is sent to the layer 2 sorter/switch in the NIC.

Step 3: Packet is sorted based on the destination MAC address; (in Fig. 2, it matches Pool/VF1).

Step 4: The NIC initiates the DMA action to transfer the packet to the VM.

Step 5: The hypervisor configures the IOMMU to perform the required address translation and the packet is moved into the VMs VF driver buffers with hardware assistance.

Step 6: The NIC posts to the hypervisor and triggers the interrupt handler in the hypervisor.

Step 7: The hypervisor injects a virtual interrupt to the VM indicating that a receiving transaction has been completed. The VMs VF driver then processes the packet.

Interrupts' necessity and overhead are critical in the virtualized network system. First, the network devices use the interrupts to notify the operating system about packet arrival. Second, the network device driver is then executed to receive the packets as described in Step 6 and Step 7. A straightforward method is to interrupt the operating system for each incoming packet. However, interrupt handling in the operating system is costly in terms of processor cycles and time [21]. Notice that the packet data can be forwarded to the VM passing through the hypervisor by the IOMMU, but the interrupts still need the VMM interventions to be injected into the VM. This procedure will incur multiple costly context switching between the VM and the hypervisor causing the main system performance bottleneck that should be mitigated.

2.2.1 VF Hardware Attributes and VF Driver

The SR-IOV shares a NIC device among the VMs and enables a VM to directly access the PCI for I/O operations through VF driver. The VF driver is a modified version of the standard 10 GE driver in which the device id is presented to the VM as part of the PCI configuration from the hypervisor to load the appropriate driver.

Link Feature Awareness. Besides the standard I/O operations (transmit and receive Ethernet packets), the VF's driver provides Link Information Awareness (LIA), such as the link status information and the link statistics information. The link status information includes *Link Speed, Link Status, and Duplex Mode*. While, the link statistics information includes *Good Packets Received Count, Good Packets Transmitted Count, Good Octets Received Count, Good Octets Transmitted Count, Multicast Packets Received Count, Function Level Reset (FLR), VLAN Tag Insertion and Checksum Insertion.*

Interrupt Coalescing. In a high performance network environment such as 10 Gigabit Ethernet, the system may experience a huge number of interrupts. It is more efficient to generate only a single interrupt for multiple packets and make the system process all incoming packets on each interrupt [22], called Interrupt Coalescing.

Moreover, the VF driver offers interrupt coalescing features in two ways. One method is delayed interrupts: the device starts a timer when the first packet arrives and no interrupt is generated until the timer expires or a certain number of packets arrive. The second method is called interrupt throttle rate control: it controls the frequency of interrupt generation. For example, with the interrupt throttle rate set to 10,000 Hz, the device will generate no more than 10,000 interrupts per second no matter how many packets are received. Interrupt coalescing is a compromise between interrupt overhead and latency [17], [23].

The interrupt coalescing techniques are often employed in high performance I/O virtualization to minimize the overhead of interrupt handling. The SR-IOV capable devices mostly support interrupt coalescing techniques such as interrupt throttle rate control and NAPI. However, the effect of SR-IOV interrupt coalescing is not satisfactory because the current solution is neither adaptive nor scalable for varying business virtualization environments.

3 SR-IOV Challenges

In this section, the virtual interrupt processing overhead is described and two major challenges in SR-IOV are revealed.

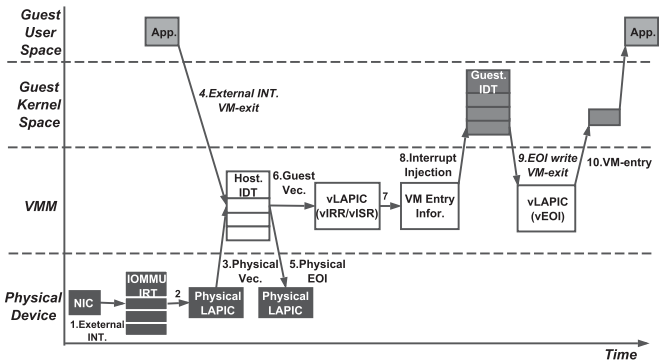


Fig. 3. Interrupt handling in x86 virtualization environment.

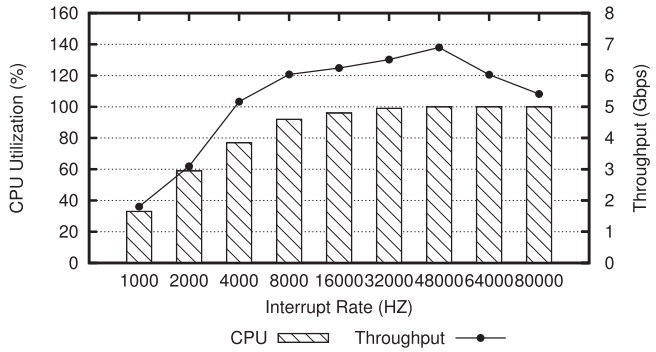


Fig. 4. Interrupt throttle rate impact on SR-IOV.

The first challenge is identifying the effects of interrupt rate on the virtualized network performance, which highlights that an adaptive interrupt control method can allow the SR-IOV to improve the throughput while limiting the interrupt overhead within an acceptable scope. The second challenge is that the SR-IOV performance is limited to ~5.6 Gbps by its single-threaded VF driver, which highlights that parallelism of the VF driver should be employed to achieve the full physical line rate.

3.1 Interrupt Overhead in Virtualized Networking

A single interrupt from the assigned devices to the VMs has to go through a long path as shown in Fig. 3. It is known that the interrupt handling is much more expensive in the virtualization environment than in the bare-metal environment [2], [16], [24]. The interrupt processing latency has been evaluated in [2] and concluded that the interrupt invocation latency varied largely due to the complicated interrupt delivery procedure in the virtualized environment.

When the external interrupt to the VMs arrives (*Step 1*), it first hits the IOMMU in the Intel Chipset, where VT-d (configured by the VMM) performs the required interrupt remapping (*Step 2*). Limited by existing x86 architecture with Intel VT-x, the VMM intercepts the interrupt (*Step 3*) and causes the VM exit events to inject the interrupts into the VMs (*Step 4*). The VM exit caused by the external interrupt is normally delivered through the host Interrupt Descriptor Table (IDT) initializing the corresponding interrupt handler. Once the physical End of Interrupt (EOI) generated (*Step 5*), the VMM maps the host vector to appropriate guest vector (*Step 6*), updates the states of the virtual Interrupt Request Register (vIRR) and the virtual In-Service Register (vISR) in the virtual local interrupt controller (vLAPIC) of destination processor and sets up the VM-entry interrupt information field in the guest controlling-VMCS (*Step 7*). Finally, the VMM uses a VM-entry event injection to deliver the virtual interrupt to the VMs (*Step 8*). The VMs in turn detect this interrupt and invoke the corresponding interrupt handler in the guest IDT. Once the processing is completed, the VM produces interrupt completion signal by performing an EOI write operation to its vLAPIC maintained by the VMM, causing another VM exit (*Step 9*) because only the VMM can perform privileged operations. Then the VMM updates the state of the virtual EOI register in VMs' vLAPIC and resumes the VMs (Step 10). A single interrupt can experience multiple rounds of trap-and-emulation [25] and the system efficiency can be seriously harmed by frequent context switching and cache pollution.

3.2 Challenge 1: Optimizing the Interrupt Rate

As mentioned above, the interrupt handling introduces high overhead in virtualized networking, but they are important for responsiveness of the system to new incoming packets. In guest domains, the VFs store the incoming packets directly in the owner VM's memory and generate interrupts to notify the VM of packet arrival. Then the VF driver schedules a NAPI polling to process the incoming packets.

Although interrupts are critical for the SR-IOV, but their frequencies have double influence on SR-IOV performance. The VF driver relies on the interrupts to process the incoming packets. If interrupts are not generated in time for incoming packets, the VF driver will not process these packets resulting in long delay or packet losses. The result reflected in the performance will be low throughput and low CPU utilization. On the other hand, a high interrupt rate will introduce high CPU overhead especially in a virtualization environment. Thus, there may be insufficient CPU resources to process the incoming packets, but the reflected result will be low throughput and high CPU utilization.

The influence of the interrupt rate is demonstrated using experiment. The testbed configuration is described in Section 5.1. The VF supports interrupt throttle rate control and the VF driver has a parameter *InterruptThrottleRate* to control the maximum number of interrupts per second. The default value of *InterruptThrottleRate* is 8,000. The impact on the SR-IOV is tested with interrupt rates of 1,000, 2,000, 4,000, 8,000, 16,000, 32,000, 48,000, 64,000 and 80,000. Fig. 4 shows that when the interrupt rate is less than 48,000, the throughput and the CPU utilization are increasing with increase in the rate. However, the throughput decreases when the interrupt rate is over 48,000 and the CPU utilization is 100 percent. The experimental results conform the analysis.

In conclusion, identifying the optimized interrupt rate is ultimately important. The optimized interrupt rate is expected to both maximize the network throughput and minimize the interrupt overhead. The optimized interrupt rate shown in Fig. 4 is approximately 48,000. The system resources are fully utilized with acceptable interrupt overhead. However, the optimized interrupt rate varies with the networking parameters such as the bandwidth and the packet size. Thus, identifying the optimized interrupt rate remains a challenge for the SR-IOV.

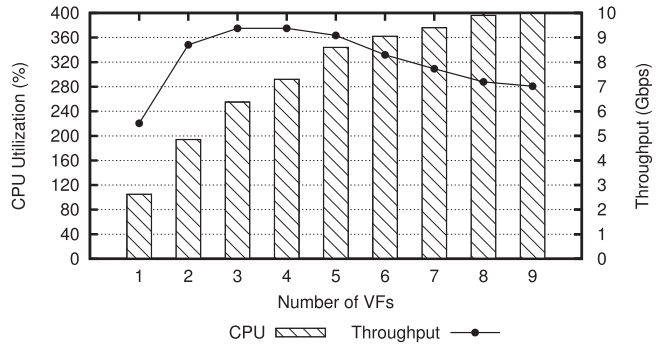


Fig. 5. SR-IOV VF scalability.

3.3 Challenge 2: Achieving Line-Rate Throughput

As discussed above, interrupt coalescing techniques and Exit-Less Interrupt have advantages in reducing the interrupt overhead and achieving almost bare-metal performance. However, even the bare-metal performance is only ~5.6 Gbps [16], but achieving the throughput of full line rate, 10 Gbps, is necessary for fully utilizing the physical link capacity. For example, several VMs are connected to the same SR-IOV device, one of the VMs is I/O intensive and others are all CPU intensive; the I/O intensive VM may need 10 Gbps throughput to accomplish its running tasks while the CPU intensive VMs may use small network flow. Another example is that all VMs on the same SR-IOV device have burst I/O events and their burst are fully interlaced; each VM requires 10 Gbps throughput during its bursts.

The performance and the scalability of the SR-IOV VF are examined, and the results are illustrated in Fig. 5. The throughput achieves ~5.6 Gbps with 1 VF and the CPU utilization is 100 percent (1 processor core). The overall throughput is easily improved to ~10 Gbps by increasing the number of VFs.

Note that the VF driver uses a single-threaded polling mechanism that has no parallelism in SMP machines. In other words, the VF driver always runs on one CPU at a time. In Fig. 5, although each VM is configured with two virtual CPUs (VCPU) and each VCPU can run on any physical CPU, the VF driver can only use one VCPU. Since one VF driver uses only one thread to poll incoming packets, the CPU utilization can never exceed 100 percent for one VM. The overall throughput increases by increasing the number of VFs because more CPU resources can be utilized with more VFs. The throughput achieves the desired physical line rate with three and four VFs. When the VF number is greater than four, the throughput starts to decrease but the CPU utilization keeps increasing. This is because of the high interrupt rate due to increased number of VF, which is discussed in Section 3.2.

Notice that one VM can achieve line rate by assigning multiple (three or four in Fig. 5) VFs. However, this will shift the burden to the applications to receive traffics from multiple VFs, which will reduce the practical utility. This motivates the authors to parallelize the VF driver to allow it to make full use of the multi-core processor and achieve full line rate throughput (10 Gbps or future higher speed network, such as 25 and 40 Gbps).

4 Optimizations

In this section, the enhancement methods for interrupt processing on the SR-IOV architecture are presented. First, the traditional interrupt rate control method is described as the baseline. Second, the mathematical model for analyzing the SR-IOV networking performance and overhead is discussed. Based on the mathematical analysis, the two interrupt rate control approaches, the coarse-grained interrupt rate control and the adaptive interrupt rate control, are proposed. The AIR and the CGR can efficiently eliminate the redundant interrupts in the SR-IOV. The second proposed optimization is multi-threaded VF driver that allows the SR-IOV to make full use of the multi-core resources to achieve full line rate.

4.1 Baseline: Fixed Interrupt Rate

Conventional device drivers provide a default fixed interrupt rate, such as 8,000 Ints/s in Linux and 4,000 Int/s in Windows, for all I/O streaming patterns without considering the throughput and the latency performance for various packet size streams. Obviously, the interrupt rate needs a dynamic adjustment to suite the practical intended environment, providing the flexibility to achieve lower latency, higher throughput and lower CPU utilization. In this paper, this fixed interrupt rate is considered as the baseline and the performance benefits achieved by the proposed interrupt rate control methods are illustrated.

4.2 Coarse Grained Interrupt Rate Control

An adjustment algorithm of interrupt throttle rate (ITR) control, called coarse-grained interrupt rate is developed. The CGR is based on the link feature awareness provided by the VF driver, as mentioned in Section 2.2.1. The CGR algorithm first maintains an *interrupt timeslice* to record the time of the latest processed interrupts. Then, the CGR employs the software legacies of Linux *ixgbe driver*, and uses the "ring_container" struct to record the number of processed packets and the amount of processed bytes during the last interrupt timeslice (100 ms). The CGR algorithm classifies the I/O traffic streams into multiple categories according to the runtime traffic attributes monitored online. Each category is empirically assigned with a predefined ITR value in a coarse-grained manner. Notice that the driver legacy is leveraged that has the control on interrupt coalescing, but the configurations are obsolete. Here, the CGR algorithm is designed with redefined I/O traffic categories and related ITR settings as follows:

(1) *Full Bulk Traffic* is with average size packet close to full-size frames (1,200-1,500 bytes). This category can essentially reach wire-speed throughput and should be assigned with a lower interrupt rate of ITR=4KInt/s in order to reduce the CPU utilization.
(2) *Intermediate Bulk Traffic* is with medium size packets (300-1,200 bytes). This category needs a higher interrupt rate of ITR=8KInt/s to balance the CPU utilization, the latency and the throughput.
(3) *Latency Sensitive Traffic* is with small packets (64-300 bytes). For this category, a comprehensive interrupt moderation set is ITR=20K Int/s to achieve an acceptable throughput with accredited I/O latency degradation.
(4) *Latency Critical Traffic* is with the occasional arrival packets of small size but with extremely sensitiveness

to the I/O process latency that normally composes of the messages that need to be processed immediately. For this category, CGR sets ITR=100KInt/s.

The categories are classified according to the experimental experiences. The ITR settings can be customized specifically. Known that Intel has released multiple versions of ixgbe driver successively, which provided the interrupt rate control methods similar to our CGR method with normally two or three traffic classifications. These interrupt rate control configurations in the ixgbe driver for 1 GB NIC became obsolete and new configurations for 10 GB NIC has been included in the current ixgbe driver version 2016. However, these configurations become obsolete immediately in the presence of emerging 40, 100 and 200 GB NICs. These effects motivate us to propose an adaptive interrupt rate control method that is independent to NIC throughput.

4.3 Adaptive Interrupt Rate Control

4.3.1 Model of Interrupts Rate Influence

Given a typical SR-IOV mathematical model: the sender is sending packets at full line rate with a constant packet size of S. Considering the packet size as a constant seems naive, however, it is reasonable if the average packet size is used in a very short time interval as S. The interrupt rate I denotes the number of interrupts per second (configured by $InterruptThrottleRate$ register), P denotes the number of packets received in one second and up to k packets can be received on each interrupt. k is the size of the VF receiving ring and is also a constant. The throughput T stands for the total size of the packets received in a second. C is a constant that represents the CPU resources capacity of the VM, C_{pkt} represents the CPU cycles consumed for processing one packet and C_{int} denotes the CPU overhead for handling a single interrupt. Then the following simple equations are obtained

$$P \leq k \times I \tag{1}$$

$$C_{pkt} \times P + C_{int} \times I \leq C. \tag{2}$$

Eq. (1) describes an upper bound on packets received in one second. Eq. (2) states the constraint that the total CPU cycles consumed for packet processing and interrupt handling cannot exceed C. Combining Eqs. (1) and (2), P can be stated as

$$P = min\left\{k \times I, \frac{C - C_{int} \times I}{C_{pkt}}\right\}. \tag{3}$$

Eq. (3) implies that P will first increase as I increases. However, P will then decrease after it reaches its maximum value (This proves the analysis discussed in Section 3.1). Let $\hat{I}$ denote the I when P is maximum and it must fit

$$k \times \hat{I} = \frac{C - C_{int} \times \hat{I}}{C_{pkt}}. \tag{4}$$

Therefore, $\hat{I}$ can be derived as

$$\hat{I} = \frac{C}{C_{pkt} \times k + C_{int}}. \tag{5}$$

As the packet size S is assumed to be a constant, throughput T has a linear relation with the number of received packets P as shown

$$T = P \times S. \tag{6}$$

Combining Eqs. (3) and (6), the relation between interrupt rate I and throughput T is

$$T = \begin{cases} k \times I \times S & \text{if I} \leq \hat{I} \\ (C - C_{int} \times I)/C_{pkt} \times S & \text{if I} > \hat{I}. \end{cases} \tag{7}$$

Eq. (7) mathematically describes the double impact of interrupts on the throughput. While the interrupt rate I is less than $\hat{I}$, the throughput T has a positive correlation with I. In such conditions, increasing the interrupt rate can benefit the system by allowing it to process more incoming packets. Thus, both the throughput and the CPU utilization will increase. On the other hand, when the interrupt rate I grows greater than $\hat{I}$, the throughput T has a negative correlation with I. In this circumstance, the CPU is fully loaded and increasing the interrupt rate will introduce more CPU overhead. Thus, less CPU resources are available for processing the incoming packets that decreases the throughput. The result of mathematical analysis conforms the experimental result shown in Fig. 4.

4.3.2 AIR Control Algorithm

Based on authors' theory of the double impact of interrupts, it can be concluded that setting the interrupt rate to $\hat{I}$ is the best trade-off between throughput and CPU overhead, as shown in Eq. (7). The interrupt rate should not be set above $\hat{I}$ at any time. However, it should also be noticed that $\hat{I}$ is derived from the assumption that the network bandwidth is fully loaded. In practice, the network bandwidth utilization may not be full. Thus, the current bandwidth B should also be considered when deciding the interrupt rate. Eq. (7) states that when I is smaller than $\hat{I}$, T increases linearly with increase in I. It means that for a certain B, there exists in theory an I to achieve full bandwidth. The appropriate interrupt rate I' for bandwidth B is described in

$$I' = \frac{B}{k \times S}. \tag{8}$$

Also, considering the upper bound of I from Eq. (5), it can be determined

$$I = min\left\{\frac{C}{C_{pkt} \times k + C_{int}}, \frac{B}{k \times S}\right\}. \tag{9}$$

An adaptive interrupt rate control algorithm is designed based on Eq. (9). Some parameters such as C_{pkt} and C_{int} are obtained from experiments. Packet size S is calculated as the average packet size in a short time interval. Bandwidth B is estimated by the throughput in the last time interval. As the bandwidth B may vary, a positive offset is always added to I to allow higher throughput in case the B increases. The difference between the current interrupt rate and the calculated interrupt rate is also tested and the interrupt rate is only updated if the difference is larger than a threshold. The adaptive interrupt rate control algorithm is described in Algorithm 1.

TABLE 1
Symbols in AIR Control Algorithm

Symbol	Meaning
B	current bandwidth
S	average packet size
k	number of packets received on each interrupt
I	calculated interrupt rate for current network condition
$\hat{I}$	the upper bound of interrupt rate by Eq. (5)
I_{min}	the lower bound of interrupt rate by VF hardware
I_{cur}	current interrupt rate

Algorithm 1. Adaptive Interrupt Rate Control

$I \leftarrow B \div (k \times S)$
$I \leftarrow I + offset$
if $I > \hat{I}$ **then**
  $I \leftarrow \hat{I}$
else if $I < I_{min}$
  $I \leftarrow I_{min}$
end if
$I_{cur} \leftarrow getInterruptRate()$
$\delta \leftarrow |I - I_{cur}|$
if $\delta \geq threshold$ **then**
  $setInterruptRate(I)$
end if

In order to avoid any ambiguities, the symbols in Algorithm 1 are listed in Table 1.

Fig. 6 shows the architecture of the adaptive interrupt rate control for SR-IOV. Two components are added in the SR-IOV VF driver: the *Statistics Unit* and the *Interrupt Rate Controller*. The *Statistics Unit* is on the packet data path and collects the information on each packet in a certain timing interval. The *Interrupt Rate Controller* obtains the information such as the average packet size and the throughput in the last time interval from the *Statistics Unit* and then determines the appropriate interrupt rate using Algorithm 1 and sets the interrupt rate in the corresponding SR-IOV VF. Since the algorithm does not require heavy calculation and

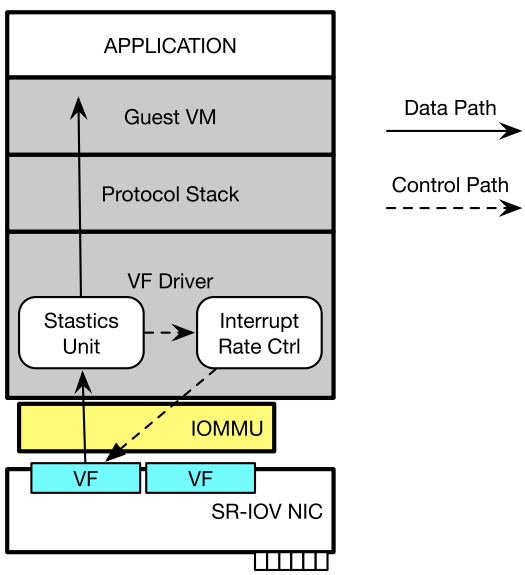


Fig. 6. Adaptive interrupt rate control architecture.

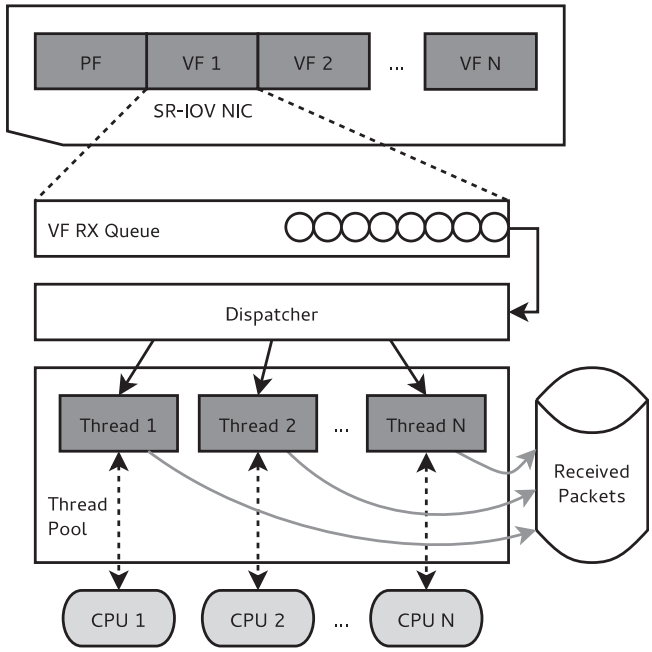


Fig. 7. Multi-threaded VF driver (MTVD) architecture.

setting the interrupt rate is as straight forward as writing a register, the overhead of adaptive interrupt rate control is trivia that is experimentally evaluated in Section 5.

4.4 Multi-Threaded VF Driver

Considering the fact that lack of parallelism is a major defect of the VF driver, the platform used in this paper is equipped with Intel Niantic 82,599 NIC that supports both SR-IOV and RSS. If available CPU cycles is represented as a variable C, then C_{min} denotes the required CPU cycles for achieving a certain throughput T. Combining Eqs. (1), (2) and (6), C_{min} can be derived as

$$C_{min} = \frac{C_{pkt} \times k + C_{int}}{k} \times \frac{T}{S}. \tag{10}$$

Eq. (10) describes the relationship between C_{min} and T. When S is fixed, C_{min} increases with increase in T. The experiments show that a single VF saturates the CPU it runs on at a throughput of about 6 Gbps, while leaving other CPUs idle. It means that when C_{min} is the available CPU cycles of one processor core, T is about 6 Gbps. C_{min} must be increased to achieve higher throughput. However, single-threaded VF driver prevents the SR-IOV from taking advantage of multi-core platforms and limits C_{min} to the capacity of one processor core. Therefore, multi-threaded VF driver is proposed to efficiently distribute the workload across multiple processor cores.

Fig. 7 illustrates the architecture of the proposed multi-threaded VF driver. The VF driver is extended with a thread pool and a dispatcher. The thread pool consists of several kernel threads with each thread bound to a CPU. The threads in the thread pool can process the incoming packets independently and concurrently. The dispatcher takes the incoming packets from the VF driver's packet buffer and dispatches them to working threads. Then the working threads process the packets and deliver them to the system

protocol stack. The dispatcher is not only responsible for assigning the packets to working threads, but also for load balancing across the working threads. The load balancing in the dispatcher is implemented using a round-robin algorithm; a packet with index i is dispatched to thread $i \bmod N$, where N is the number of working threads.

5 Evaluation

5.1 Experimental Setup

The experiment testbeds in this study consist of two IBM server machines, each equipped with 2 Intel Xeon 2.27 GHz E5607 CPU with 4 cores, VT-d support and 24 GB of RAM. One server runs a KVM hypervisor with Linux 2.6.32 kernel hosting from 1 VM to 16 VMs. Each VM runs in hardware virtual machines (HVM) with Linux 2.6.32 kernel. A single VM is configured with 2 vCPUs, 512 MB of RAM and 8 GB storage along with an Intel 82,599 Ethernet Controller Virtual Function. The other server runs as a client for generating network traffics. The two machines are connected directly through a dual-port Intel 82,599 10 Gigabit Ethernet Controller.

5.2 NetPerf Testing

The network performance is evaluated with *Netperf* benchmark in case of stream transferring with small, medium and large packets sizes of 50-byte, 512-byte and 1,472-byte, respectively. The CPU utilization is presented as a percentage of a single processor core. Note that the maximum CPU utilization is 800 percent as the hypervisor used in the platform has 8 physical cores.

5.2.1 TCP_STREAM Performance

Throughput. Fig. 8 shows the throughput performance of *TCP_STREAM*, labelled on the right y-axis, with different packet sizes. The FIR baseline is compared with CGR and AIR in different packet streams. Fig. 8a presents the throughput of TCP_Stream with 1,472-byte packet, where CGR and AIR achieve 1.22x∼2.26x and 1.31x∼2.97x higher throughput than the FIR, respectively. Similarly, Fig. 8b presents the TCP_Stream with 512-byte packet, where CGR and AIR achieve 1.13x∼1.98x and 1.11x∼2.78x higher throughput than the FIR, respectively. These performance benefits come from the adequate interrupt throttle rate adjustment and the highest performance improvements appear in more severe I/O intensive environment, such as with 16VMs. Moreover, the AIR has the best scalability since it achieves the peak throughput and stays stable with the increasing number of VMs. In contrast, both FIR and CGR obtain improved throughput when with 1VM to 8VM but both degrade significantly when with more than 8VM, as shown in Figs. 8a and 8b.

Fig. 8c shows the throughput of TCP_Stream with 50-byte packet, where CGR and AIR both have the similar performance to FIR. However, it can be seen that the FIR has almost exhausted all CPU resources due to the high packet arrival rate in the TCP_Stream with small packets. Thus, the advantages of CGR and AIR are reflected in the CPU utilization that are discussed in the following sections.

CPU Utilization. The CPU utilization presented at the left y-axis of Fig. 8 denotes the consumed computation resources of the Netperf and a 100 percent utilization means that the entire CPU core is consumed.

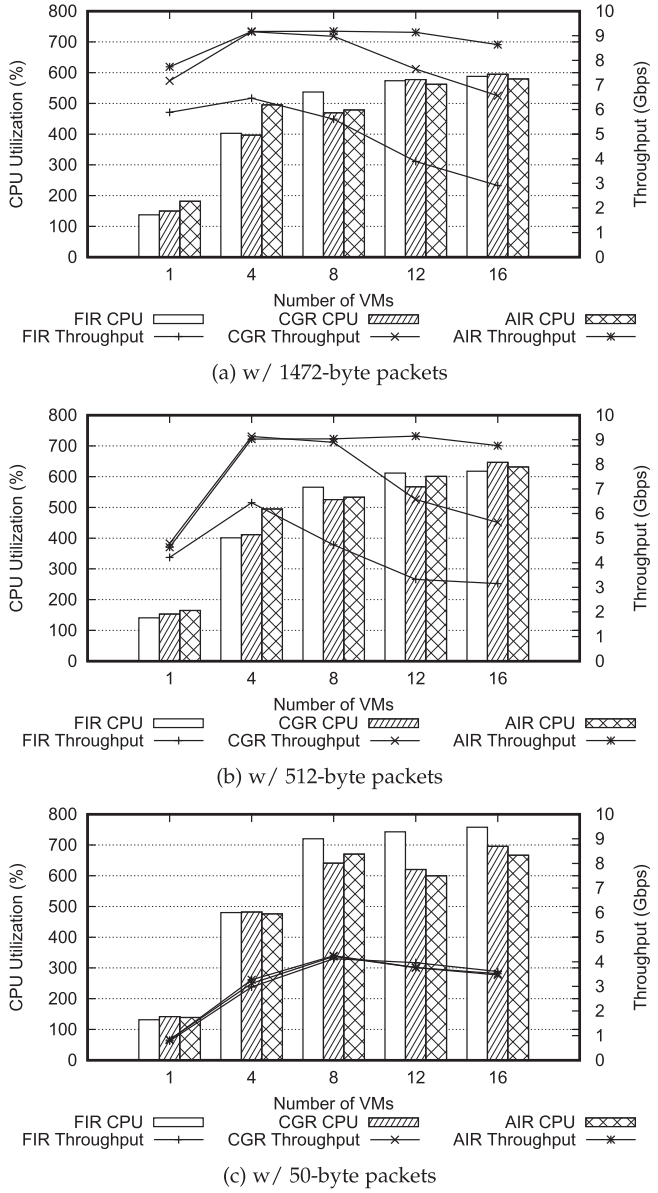
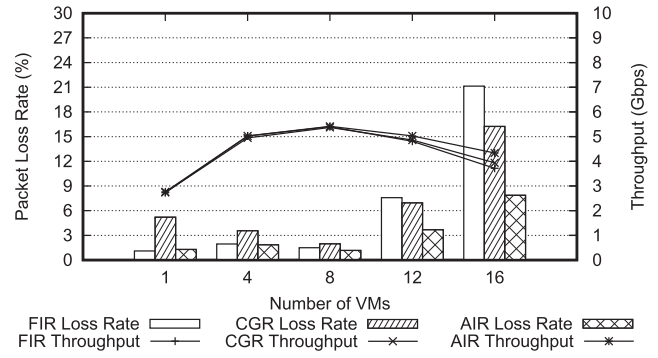


Fig. 8. TCP_Stream throughput and CPU utilization.

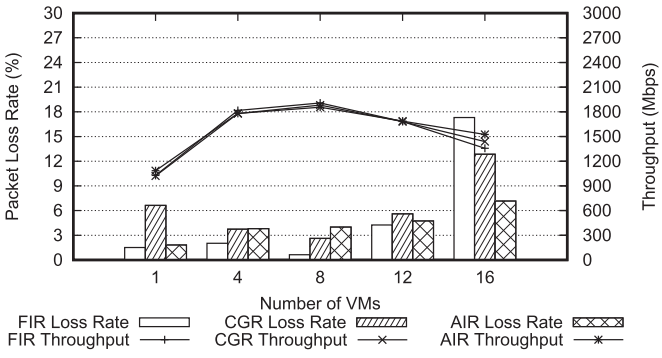
It can be observed that the CPU utilization is under 600, 646 and 758 percent for TCP_Stream with 1,472-byte packet in Fig. 8a, with 512-byte packet in Fig. 8b and with 50-byte packet in Fig. 8c, respectively. This effect shows that the smaller the TCP_Stream packet size is, the more the CPU resources are consumed.

Notice that in Fig. 8c, both the CGR and the AIR have no throughput improvement compared to the FIR, but they can largely save CPU resources. For example, when with 12VMs, the CGR and the AIR can save up to 1.23 and 1.44 cores, respectively. This effect means that the advantages of adequate interrupt rate control are reflected in efficient CPU utilization.

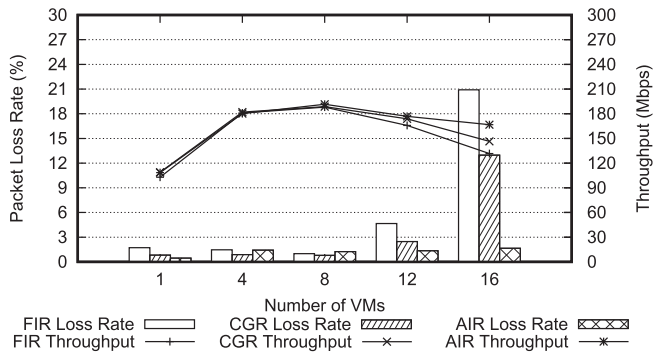
Overhead. In Figs. 8a, 8b, and 8c, the AIR consumes more CPU computation resources when VM number is small such as 1 and 4 VM. This denotes the AIR's CPU overhead for computing adequate ITR values. However, the AIR outperforms the CGR and the FIR with higher throughput



(a) w/ 1472-byte packets



(b) w/ 512-byte packets



(c) w/ 50-byte packets

Fig. 9. UDP_Stream throughput and packet loss rate.

when the VM number is more than 4 due to suitable interrupt rate setting. Therefore, it can be concluded that the adaptive interrupt rate control has acceptable overhead and the performance advantages of adequate ITR setting can easily compensate the incurred overhead.

5.2.2 UDP_STREAM Performance

The performance is also evaluated using UDP_Stream, and it is observed that both the CGR and the AIR have no considerable advantages over the throughput but both have significant advantages in reducing the *packet loss rate*. This is because the UDP does not have transmission control and the Ethernet layer flow control will be triggered in case of packet loss.

Packet Loss Rate (PLR) Reduction illustrated on the left side y-axis in Fig. 9 is defined according to the difference of packet number between the sending side and the receiving side. It shows that the *packet loss rate* for FIR algorithm increases approximately in an exponential manner along with the increasing number of VMs. With high contention of CPU cores, each VM is allocated with a handful of resources. Thus, more UDP packets are unable to be processed in time resulting in losing packets.

As shown in Figs. 9a and 9b, the *PLR* is very low when the VMs are less than 8 because all the arrival packets can be processed by the VMs. In all cases, the AIR never exceeds 8 percent in Figs. 9a and 9b. However, when the number of VMs is 16, the *PLR* of FIR reaches up to 21.13 percent as shown in Fig. 9a. The CGR and the AIR both have much lower *PLR* compared to the FIR because they both try to configure an appropriate ITR value to promptly process the incoming packets. Moreover, the contention of CPU cores aggravates the problem with large number of VMs causing rapid grow in the *PLR*. The performance advantages of AIR become obvious when the number of VMs is 16. For example, the *PLR* of AIR is one tenth of the FIR (2 versus 21 percent) and one sixth of the CGR (2 versus 13 percent) when the number of VMs is 16. It can be noticed in Fig. 9 that the highest throughput and the lowest loss rate are achieved when hosting 8VMs. This is because the testbed server has 8 cores and each VM is allocated one physical exclusively. Higher number of VMs will incur context switch overhead for the vCPU multiplexing, while less number of VMs will incur a higher network pressure to each VM resulting in degraded throughput and a higher loss rate.

It can be summarized that the stationary and inappropriate ITR setting of FIR results in throughput degradation and *PLR* increase. The improvements of CGR and AIR are obtained by efficient adjustment of ITR value, so that each VM can timely process the incoming packets.

5.3 WebBench Testing

The WebBench benchmark is used to evaluate the performance in WWW or proxy servers for handling massive concurrent clients' requests. Fig. 10 presents the performance in terms of *throughput* and *total failed clients*. The throughput and the number of total failed clients denote the total page requests processed by all VMs per minute and the total requests that fail to be accomplished, respectively. The number of clients that concurrently send the requests to each VM is varied as 100, 1,000 and 2,000.

Throughput. Figs. 10a, 10b, and 10c show that the AIR always achieves the highest throughput, the CGR performs secondarily, and the FIR has the lowest throughput. The results prove the efficiency that the AIR dynamically configures a proper ITR value to improve the WebBench throughput rather than merely sets a coarse-grained value or a constant ITR value. The most significant throughput improvement is achieved when there are 2,000 clients, CGR and AIR obtain the improvement up to 13.58 and 21.24 percent compared to the FIR, as shown in Fig. 10c.

Total Failed Clients. Figs. 10a, 10b, and 10c show that there are no failed clients when 100 concurrent clients send request to each VM, while the failed clients appear with 1,000 and 2,000 clients due to the higher workload pressure. Fig. 10a shows that the CGR and the AIR both can process all requests without failed clients. However, the FIR has failed clients when 8VMs are hosted on the 8-core server. Fig. 10c illustrates that the failed clients numbers are

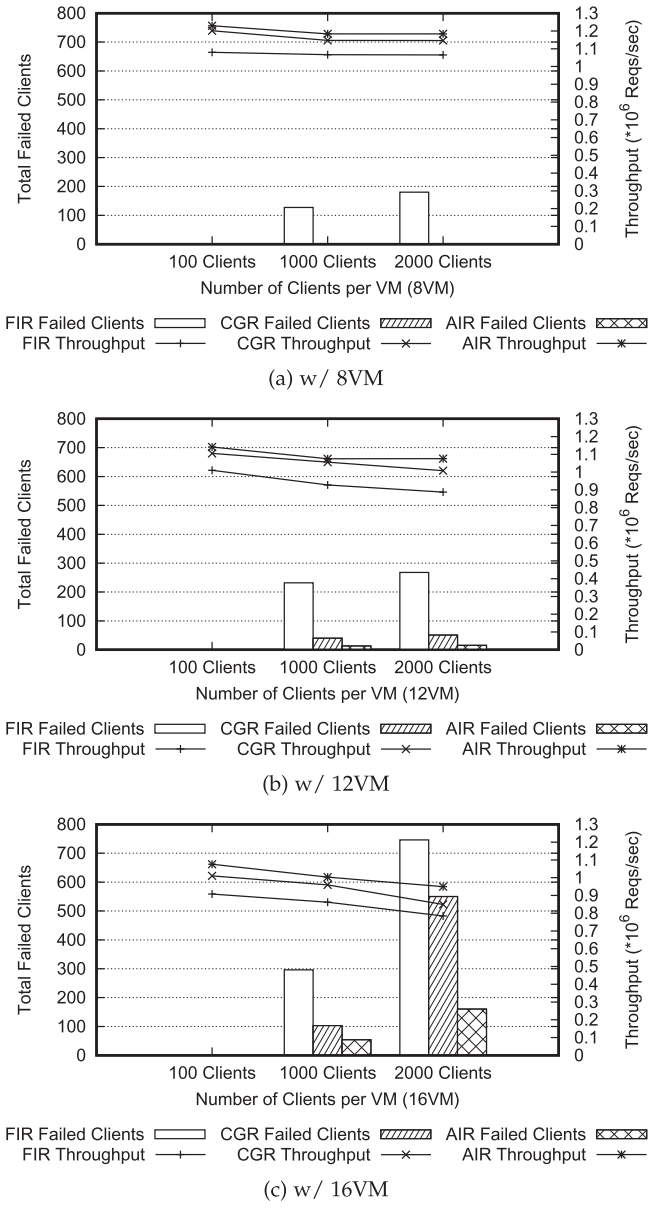


Fig. 10. WebBench in throughput and failed clients.

remarkable when 16 VMs contend CPU resource, but the AIR and the CGR are still able to significantly reduce the number of failed clients in comparison with the FIR. For example, where each VM processes 2,000 concurrent clients, the AIR and the CGR reduce the total failed clients of the co-hosted 16 VMs down to 160 and 550, respectively, compared to 746 total failed clients caused by the FIR.

5.4 Apache Bench Testing

The authors further evaluate using Apache Bench (ab) as a more practical HTTP server, since ab is a specialized measurement tool that represents the server's capacity of processing requests. Fig. 11 shows the evaluation of the completion times (time taken for test) for servicing a varied number of requests (1K∼10K). A concurrent client number is configured as high as 500 resulting in an approximate value among the adjoining number of requests. It is clear from the figure that the time taken for accomplishing the request by the FIR quickly increases with the increasing

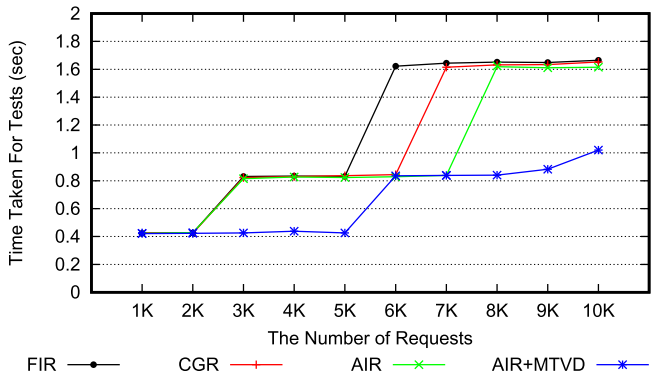


Fig. 11. Apache Bench performance in requests processing.

number of requests, while CGR and AIR both perform slightly better than the FIR and the AIR+MTVD shows a slowest pace of increase. This proves that the AIR+MTVD has the highest capacity to treat the pressure of client requests. Hence, A realistic system (web applications) can substantially benefit from an adaptive interrupt rate control method and multi-threaded driver.

5.5 Multi-Threaded VF Driver Influence

This section shows the performance enhancement obtained with the multi-threaded VF driver using the TCP_Stream and the UDP_Stream throughput tests. The Netperf streaming packet size is varied as 50-byte, 512-byte and 1,472-byte. The multi-threaded VF driver takes more use of cores resources to enhance the capacity of processing the arrival packets. Fig. 12 shows that the throughput can be significantly improved with additional CPU resource consumption for 4 co-hosted VMs each of which is configured with 2 vCPUs.

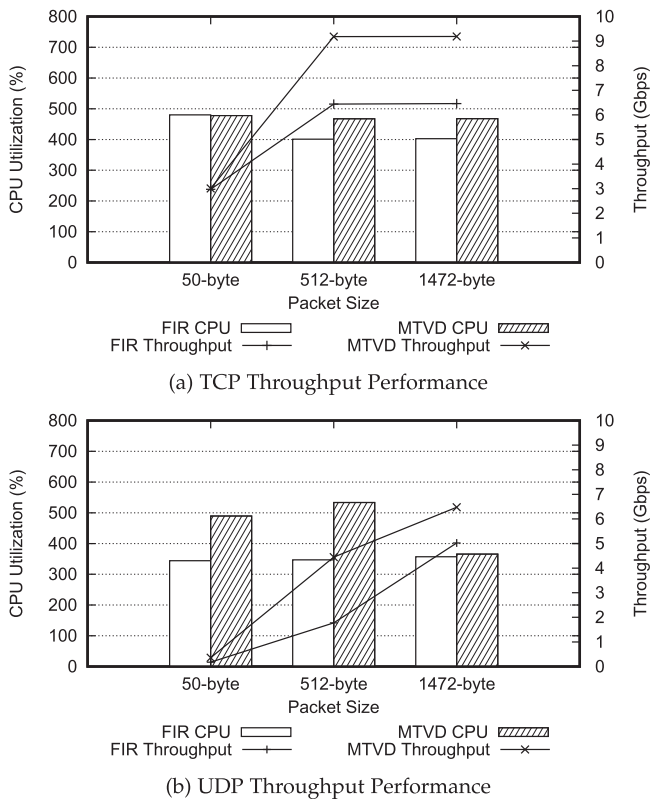


Fig. 12. Multi-threaded VF driver (MTVD) enhancement.

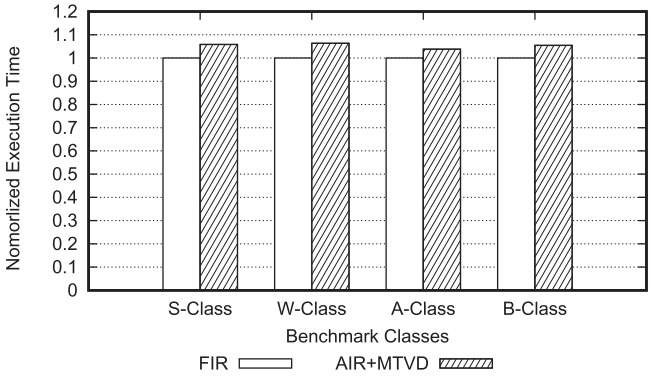


Fig. 13. Overhead impact for CPU-intensive VMs.

Fig. 12a presents the TCP_Stream performance with different packet sizes. For small packets, the MTVD has similar performance as the FIR, because the sending side achieves the bottleneck and all arrival packets are processed. For TCP_Stream with medium and large size packets, the MTVD achieves up to 42.21 percent higher throughput with an additional 64.86 percent CPU utilization than the FIR.

Fig. 12b presents the results of UDP throughput along with CPU utilization. For UDP_Stream with 1,472-byte packets, the MTVD consumes an additional 8.76 percent CPU resources to obtain 28.89 percent higher throughput than the FIR. For UDP_Stream with 512-byte packets, the MTVD improves the throughput by 102.93 percent with an additional 145.54 percent CPU utilization. In other words, the MTVD achieves 2.03x performance with additional 1.46 cores consumption. It should be noticed that the UDP throughput merely reaches 367.1 Mbps that is far less than 10 Gbps, because the network bottleneck is on the sending side transmitting fewer UDP packets. The evaluation shows that the throughput of the TCP_Stream and the UDP_Stream can be significantly improved by the MTVD with additional CPU resource consumptions. Therefore, the MTVD is suitable for the situation where a single VM can hardly process more incoming packets so that a single thread soon becomes a bottleneck.

5.6 Overhead of MTVD and AIR

Notice that the MTVD extends the VF driver in multi-threaded manner and achieves improved performance for an I/O-intensive VM. The NAS Parallel Benchmark (NPB v3.3.1) is executed as computation-intensive workload to illustrate the overhead of MTVD and AIR. The Fourier Transform (FT) testing of the NPB benchmark suite is tested, as a MPI version running in single-threaded mode with four scale test set classes: S, W, A and B. Class S is small and used for quick test purposes, Class W represents workstation size, Classes A and B are standard test problems, and Class B is 4x larger in size than Class A. These test sets have already achieved the upper data capacity of our our VM with 500M memory. The execution time of a VM running NPB FT benchmark is evaluated and the difference between a baseline VM with FIR and a VM with MTVD+AIR is compared to show their overhead impacts on computation-intensive applications. The results are normalized in Fig. 13, and it is shown that the AIR+MTVD costs at most 6 percent additional time compared with the FIR in all tested Classes. Under more realistic environment where some VMs are CPU-intensive, the MTVD achieves considerably better throughput while still handling the CPU-intensive tasks timely.

5.7 Summary

The AIR outperforms the CGR and the FIR in VM contention and network congestion environments. The evaluation of *Netperf* shows that the CGR and the AIR both can essentially improve the throughput and *packet loss rate* while the CPU utilization sometimes can be saved or almost remains the same with the FIR. With more VM adding into the system, the throughput begins to drop and the CPU utilization starts to level off. It is found that the zenith of throughput is approximated at 8VMs due to platform specific configuration. The *WebBench* verifies the improvement in speed and confirms the reduction of the number of failed clients. Apache bench validates that the AIR and the MTVD benefit on the generic HTTP server. Finally, IS of the NPB benchmark proves that the MTVD+AIR algorithm can effectively improve the throughput with trivial CPU overhead.

6 Related Work

This section discusses the previous researches on optimizing I/O virtualization.

I/O virtualization has always been on the list of concerns of researcher working on virtualization since the invention of virtual machines. High performance network virtualization is extremely important in cloud computing as all data flows through a high performance network such as 10 GE. However, network virtualization suffers from high overhead and low performance. In order to resolve these issues, many network virtualization optimizations have been proposed. Some of these optimizations address the performance bottleneck in traditional software-based I/O virtualization model such as Xen frontend-backend I/O para-virtualization. Some studies have found that the VMM schedulers such as the Xen credit scheduler are not optimized for I/O tasks and proposed efficient optimizations for the schedulers to achieve better I/O performance. Others use hardware-based I/O virtualization techniques such as self-virtualized devices and direct I/O.

6.1 Software-Based I/O Virtualization Model

Ahmad et al. designed and implemented a virtual interrupt coalescing (VIC) scheme for virtual SCSI hardware controllers for VMware ESX. Their results showed that the performance can be improved by up to 18 percent in micro benchmarks and up to 5 percent in TPC-C [23].

Gordon et al. reduced the overhead of interrupt handling in virtual machines by Exit-Less Interrupts. The ELI is a software approach to directly and securely handle interrupts by using the guest virtual machines. The ELI allows guest to reach 97-100 percent of bare-metal performance [16]. However, the ELI suffered from scalability problem and therefore the authors further proposed ELVIS [13] method for multiple guests to alleviate the overhead of paravirtual I/O by running host functionality on dedicated cores and separating them from the guest cores. While ELI and ELVIS reduced the overhead of each interrupt handling, the AIR in this paper is proposed to further reduce

the number of interrupts. In other words, it is the same problem that is solved from different aspects and the proposed MTVD can exceed the performance of bare-metal to effectively employ the line rate speed. Altogether, the presented work can be an integrated and adaptive solution for I/O virtualization.

Deri et al. explored the increased parallelism of commodity hardware and proposed the Threaded NAPI that allowed the packet capturing to scale up with the number of cores [18]. Their solution relied on special hardware features such as Receive Side Scaling and multiple RX/TX queues. The TNAPI requires more optimization, which is in End-of-life replacing by advanced direct network access (DNA) technologies. Virtual machines with direct device assignment cannot provide RSS or multi-queue supporting in each assigned device (why only multiple VFs can achieve ~10 Gbps). On the contrary, the proposed MTVD is a software-only approach that is more applicable for virtualized environments for each VM network connection.

Menon et al. optimized the Xen driver domain I/O virtualization model by avoiding the remapping operations on the data transmit and receive paths. They also made it possible for the guest OS to use advanced virtual memory features such as superpages and global page remapping [26]. Liao et al. examined the negative effect of virtualization in multicore platforms with 10 GE networking and proposed two optimizations, cache-aware scheduling and I/O VCPU favored scheduling [27]. Dong et al. proposed two optimizations for network virtualization; interrupt coalescing technique was employed to reduce the CPU overhead while virtual receive side scaling was used to allow the Xen driver domain to make full use of multiple processor cores [14]. Wang et al. proposed XenLoop, a transparent high performance inter-VM network channel, to improve the inter-VM network performance. XenLoop makes the inter-VM communications extremely efficient through shared memory [28]. Liu et al. proposed a Virtualization Polling Engine (VPE) that used the dedicated CPU cores to accelerate the I/O virtualization with dedicated polling threads. The experiments showed that the VPE was able to significantly reduce the I/O virtualization overhead and achieve performance close to hardware-based approaches [29]. Normally, software-based schemes will incur a significant system overhead with high CPU utilization. This overhead can be eliminated by using dedicated I/O core, also named as "sidecore" model. However, the dedicated sidecores on each host might be wasteful when I/O activity is low, or it might not provide enough computational power when I/O activity is high [29], [30]. Note that the VPE used an event-driven polling thread and the events can be timer interrupt. It is a similar problem of dynamic interrupt rate control that has been resolved by employing the Interrupt Throttle Rate register on the hardware NIC in this paper. In what follows, the hardware-based I/O virtualization techniques are described.

6.2 Hardware-Based I/O Virtualization Techniques

Hardware-based I/O virtualization has also a long research history. As far back as 1989, Borden et al. described a method to implement direct I/O that allowed multiple operating systems to run on one processor complex [31]. Shivam et al. proposed a completely new zero-copy, OS-bypass messaging layer called Ethernet Message Passing (EMP). The EMP allowed the Gigabit Ethernet on Alteon NIC to process the entire protocol stack at the NIC and achieved excellent performance (latency of 23 s, and throughput of 880 Mbps) [32]. Liu et al. suggested that the VMM appeared to be a performance bottleneck for system with high I/O demands. They extended the idea of OS-bypass I/O and proposed a new device virtualization model called VMM-bypass I/O. The VMM-bypass I/O allowed the VMs to perform time-critical I/O operations directly in guest VMs without involving the VMM. The VMM-bypass I/O can significantly improve the I/O performance without sacrificing safety or isolation [33]. Raj et al. proposed self-virtualized devices that improved I/O performance by offloading the select virtualization functionality onto the device [7]. The self-virtualized devices provide virtual interfaces to VMs for an underlying physical device. Willmann et al. explored the performance and safety tradeoffs of strategies for using direct I/O techniques [34]. Virtual Machine Device Queues (VMDq) is another method to offload the network I/O data processing from software to hardware. VMDq capable devices provide multiple transmit and receive queues, each dedicated to a VM. The device hardware could demultiplex the packets to VMs by putting them in queues [35], [36].

The SR-IOV moves the hardware-based I/O virtualization a step further by providing multiple virtual functions (VF) with a single physical function. Each VM can be assigned with one or more VFs and can access the VFs directly. The SR-IOV utilizes the direct I/O and achieves almost native performance [9], [11], [37]. Li et al. proposed VM-proof XRC to eliminate the scalability gap between virtualized and native environments. Their evaluation showed that the modern high-speed interconnection networks could get the same raw performance and scalability as in native environments with VM-proof XRC [38]. This paper considers the SR-IOV hardware assisted virtualization technology and proposes two flexible interrupt rate control schemes (CGR and AIR) using the networking statistic information and the ITR control register of VF on the SR-IOV NIC. The proposed methods follow author's previously proposed software-based adaptive scalable interrupt rate optimization method [39] and leverage further the statistic information provided on the SR-IOV VF device register to eliminate the system overhead of network status monitoring.

6.3 Optimizing VMM Schedulers for I/O

Researchers at the Beihang University have divided the processor cores into three subsets: driver cores, fast-tick cores and general cores. Each subset employs a specific scheduling strategy to meet the requirements of different tasks [40].

Cherkasova et al. compared the three CPU schedulers for Xen: BVT, SEDF and the credit scheduler. They analyzed the impact of the choice of the scheduler and its parameters on the application performance and discussed the challenges in estimating the application resource requirements in virtualized environments [41]. Ongaro et al. were the first to study the impact of the VMM scheduler. They found that the traditional VMM schedulers mainly focused on sharing the processor among the virtual machines while I/O tasks may not be scheduled in time [42]. Gupta et al. accurately

measured per-VM resource consumption using XenMon and proposed SEDF-DC to enforce performance isolation for a variety of workloads and configurations [43]. A task-aware virtual machine scheduling mechanism was proposed by Kim et al. [44]. This scheme schedules the I/O-bound tasks selectively to promptly handle the incoming events. Govindan et al. addressed a key shortcoming in the existing VMMs that CPU schedulers are agnostic to the communication behavior of modern multi-tier applications. They developed a new communication-aware CPU scheduling algorithm to alleviate this problem [45].

7 Conclusion

This paper mainly focuses on the performance enhancement and optimization in adaptivity and scalability for SR-IOV based high speed network virtualization. First, the two challenges in the SR-IOV caused by excessive interrupts and single-threaded VF driver are examined. Then, new optimizations are proposed to address these challenges. The first optimization method is the interrupt rate control that adaptively and automatically adjusts the interrupt rate based on a mathematical model. The second method is the multi-threaded VF driver that efficiently distributes the workload across the processors. The proposed optimizations are implemented in the VF driver and detailed experiments are presented to evaluate and demonstrate their benefits. The comprehensive experimental results validate that the adaptive interrupt rate control or even the coarse-grained interrupt rate control can save CPU utilization while achieving higher throughput than the traditional fixed interrupt rate processing scheme. At the same time, the proposed multi-threaded VF driver can improve the SR-IOV performance with additional CPU resource consumption to achieve high scalability. The planned future work will extend the proposed adaptive interrupt rate control scheme on Posted-interrupt technology that can significantly reduce the overhead in interrupt delivery and injection.

Acknowledgments

This work was supported in part by the National Key Research & Development Program of China (2016YFB1000500), and National Natural Science Funds for Distinguished Young Scholar No.61525204.

References

[1] K. Bakshi and C. Hill, "Cloud network and I/O virtualization," *Encyclopedia Cloud Comput.*, vol. 9, no. 102, 2016.

[2] C.-C. Tu, M. Ferdman, C.-T. Lee, and T.-C. Chiueh, "A comprehensive implementation and evaluation of direct interrupt delivery," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2015, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/2731186.2731189

[3] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proc. 1st ACM/USENIX Int. Conf. Virtual Execution Environ.*, 2005, pp. 13–23.

[4] X. Xu and B. Davda, "SRVM: Hypervisor support for live migration with passthrough SR-IOV network devices," in *Proc. 12th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2016, pp. 65–77.

[5] P. Barham, et al., "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 164–177.

[6] K. Avi, "KVM : The Linux virtual machine monitor," *Proc. Linux Symp.*, vol. 1, pp. 225–230, 2007.

[7] H. Raj and K. Schwan, "High performance and scalable I/O virtualization via self-virtualized devices," in *Proc. 16th Int. Symp. High Perform. Distrib. Comput.*, 2007, pp. 179–188.

[8] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the xen virtual machine monitor," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, p. 24.

[9] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *Proc. IEEE 16th Int. Symp. High Perform. Comput. Archit.*, 2010, pp. 1–10.

[10] D. Münch, M. Paulitsch, O. Hanka, and A. Herkersdorf, "MPIOV: Scaling hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems," in *Proc. Des. Autom. Test Eur. Conf. Exhib*, 2015, pp. 579–584.

[11] G. K. Lockwood, M. Tatineni, and R. Wagner, "SR-IOV: Performance benefits for virtualized interconnects," in *Proc. Annu. Conf. Extreme Sci. Eng. Discovery Environ.*, 2014, Art. no. 47.

[12] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.

[13] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual I/O system," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2013, pp. 231–242.

[14] Y. Dong, D. Xu, Y. Zhang, and G. Liao, "Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2011, pp. 26–34.

[15] J. Li, R. Ma, H. Guan, and D. S. Wei, "vINT: Hardware-assisted virtual interrupt remapping for SMP VM with scheduling awareness," in *Proc. IEEE 7th Int. Conf. Cloud Comput. Technol. Sci.*, 2015, pp. 234–241.

[16] A. Gordon, et al., "ELI: Bare-metal performance for I/O virtualization," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 411–422.

[17] J. H. Salim, "When NAPI comes to town," in *Proc. Linux Conf.*, Swansea, UK, Aug. 2005.

[18] L. Deri and F. Fusco, "Exploiting commodity multi-core systems for network traffic analysis," 2009. [Online]. Available: http://luca.ntop.org/MulticorePacketCapture.pdf

[19] N. Amit, M. Ben-Yehuda, and A. Gordon, "vIOMMU: Efficient IOMMU emulation," in *Proc. USENIX Annu. Tech. Conf.*, 2011, p. 6.

[20] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2010, pp. 1–12.

[21] M. Ben-Yehuda, et al., "The turtles project: Design and implementation of nested virtualization," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 423–436.

[22] K. Salah and A. Qahtan, "Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme," *Comput. Commun.*, vol. 32, no. 1, pp. 179–188, 2009.

[23] I. Ahmad, A. Gulati, and A. Mashtizadeh, "vIC: Interrupt coalescing for virtual machine storage device IO," presented at *USENIX Annu. Tech. Conf.*, Portland, OR, USA, 2011.

[24] T. Kun, Y. Dong, X. Mi, and H. Guan, "sEBP: Event based polling for efficient I/O virtualization," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 135–143.

[25] G. Popek and R. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[26] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in xen," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2006, p. 2.

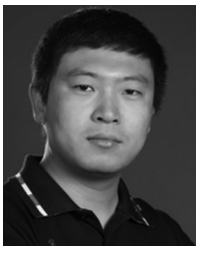
[27] G. Liao, D. Guo, L. Bhuyan, and S. R. King, "Software techniques to improve virtualized I/O performance on multi-core systems," in *Proc. 4th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2008, pp. 161–170.

[28] J. Wang, K.-L. Wright, and K. Gopalan, "XenLoop: A transparent high performance inter-VM network loopback," in *Proc. 17th Int. Symp. High Perform. Distrib. Comput.*, 2008, pp. 109–118.

[29] J. Liu and B. Abali, "Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization," in *Proc. 23rd Int. Conf. Supercomputing*, 2009, pp. 225–234.

[30] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir, "Paravirtual remote I/O," *SIGOPS Operating Syst. Rev.*, vol. 50, no. 2, pp. 49–65, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2954680.2872378

[31] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk, "Multiple operating systems on one processor complex," *IBM Syst. J.*, vol. 28, no. 1, pp. 104–123, 1989.
[32] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing," in *Proc. ACM/IEEE 2001 Conf. Supercomputing*, Nov. 2001, Art. no. 49.
[33] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance VMM-bypass I/O in virtual machines," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2006, p. 3.
[34] P. Willmann, S. Rixner, and A. L. Cox, "Protection strategies for direct access to virtualized I/O devices," in *Proc. USENIX Annu. Tech. Conf. Annu. Tech. Conf.*, 2008, pp. 15–28.
[35] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for I/O virtualization," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 29–42.
[36] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner, "Achieving 10 gb/s using safe and transparent network interface virtualization," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2009, pp. 61–70.
[37] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, "Towards high-quality I/O virtualization," in *Proc. Israeli Exp. Syst. Conf.*, 2009, pp. 12:1–12:8.
[38] B. Li, Z. Huo, P. Zhang, and D. Meng, "Virtualizing modern high-speed interconnection networks with performance and scalability," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2010, pp. 107–115.
[39] Z. Huang, R. Ma, J. Li, Z. Chang, and H. Guan, "Adaptive and scalable optimizations for high performance SR-IOV," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 459–467.
[40] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia, "I/O scheduling model of virtual machine based on multi-core dynamic partitioning," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 142–154.
[41] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, Sep. 2007.
[42] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2008, pp. 1–10.
[43] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2006, pp. 342–362.
[44] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2009, pp. 101–110.
[45] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: Communication-aware CPU scheduling for consolidated xen-based hosting platforms," in *Proc. 3rd Int. Conf. Virtual Execution Environ.*, 2007, pp. 126–136.



Jian Li received the PhD degree in computer science from the Institut National Polytechnique de Lorraine (INPL)-Nancy, France, in 2007. He is an associate professor in the School of Software, Shanghai Jiao Tong University. His research interests include virtualization, cyber-physical system, and cloud computing.

Shuai Xue is working toward the master's degree in the School of Software, Shanghai Jiao Tong University, China. His main research interests include virtualization and cloud computing.

Wang Zhang is working toward the master's degree in the School of Software, Shanghai Jiao Tong University, China. Her main research interests include virtualization and cloud computing.

Ruhui Ma received the BS and MS degrees from the School of Information and Engineering, Jiangnan University, China, in 2006 and 2008, respectively, and the PhD degree in computer science from Shanghai Jiao Tong University, China, in 2011. His main research interests include virtual machines, computer architecture, and compiling.

Zhengwei Qi received the BEng and MEng degrees from Northwestern Polytechnical University, in 1999 and 2002, respectively, and the PhD degree from Shanghai Jiao Tong University, in 2005. He is a professor in the School of Software, Shanghai Jiao Tong University. His research interests include cloud computing, virtualization, and program analysis.

Haibing Guan received the PhD degree in computer science from Tongji University, China, in 1999. He is a full professor in Department of Computer Science, Shanghai Jiao Tong University, China. His current research interests include but are not limited to computer architecture, compiling, virtualization, and hardware/software co-design. He is a member of the ACM, the IEEE, and the CCF.