High Performance Network Virtualization with SR-IOV

Yaozu Dong*, Xiaowei Yang*, Xiaoyong Li†, Jianhui Li*, Kun Tian*, Haibing Guan†[#]

*Intel China Software Center,
Shanghai, P.R.China
* {eddie.dong, xiaowei.yang, jian.hui.li, kevin.tian}@intel.com

† Shanghai Jiao Tong University,
Shanghai, P.R.China
† {xiaoyongli, hbguan}@sjtu.edu.cn

Abstract—**Virtualization poses new challenges to I/O performance. The single-root I/O virtualization (SR-IOV) standard allows an I/O device to be shared by multiple Virtual Machines (VMs), without losing runtime performance. We propose a generic virtualization architecture for SR-IOV devices, which can be implemented on multiple Virtual Machine Monitors (VMMs). With the support of our architecture, the SR-IOV device driver is highly portable and agnostic of underlying VMM. Based on our first implementation of network device driver, we applied several optimizations to reduce virtualization overhead. Then, we carried out comprehensive experiments to evaluate SR-IOV performance and compare it with paravirtualized network driver. The results show SR-IOV can achieve line rate (9.48Gbps) and scale network up to 60 VMs at the cost of only 1.76% additional CPU overhead per VM, without sacrificing throughput. It has better throughout, scalability, and lower CPU utilization than paravirtualization.**

Keywords: Virtualization; Virtual Machine; SR-IOV; Xen;

I. INTRODUCTION

I/O performance is critical to high performance computer systems. With the rapid development of multi-core technology and Non-Uniform Memory Architecture, CPU computing capabilities and memory capacities keep increasing according to Moore's Law, but I/O performance suffers from both slow PCI Express (PCIe) [14] link and hardware scalability limitation, such as the limit on the number of PCIe slots. I/O intensive servers and clients may waste CPU cycles waiting for available data or spinning on idle cycles. This reduces overall system performance and scalability.

Virtualization allows multiple OSes to share a single physical interface, to maximize the utilization of computer system resources, such as I/O devices. An additional software layer, named Virtual Machine Monitor (VMM) or hypervisor [18], is introduced to provide the illusion of Virtual Machines (VMs), on top of which each OS assumes owning resources exclusively. There are two approaches to enable virtualization. Paravirtualization (PV) [27] requires OS modification to work cooperatively with VMM. Full virtualization requires no modification, using hardware supports like Intel® Virtualization Technology [24]. Xen [1] is an open source VMM which supports both paravirtualization and full virtualization. It runs service OS in a privileged domain (domain 0) and multiple guest OSes in the guest domain.

[#] Corresponding Author

I/O performance and scalability needs improvement to effectively handle virtualization. When a guest accesses the I/O device, VMM needs to intervene in the data processing to securely share the underlying physical device. The VMM intervention leads to additional I/O overhead for a guest OS. Existing solutions, such as Xen split device driver [4], also known as PV driver, suffer from VMM intervention overhead, due to packet copy [21][16]. The virtualization overhead could saturate the CPU in a high throughput situation, such as 10 Gigabit Ethernet, and thus impair the overall system performance.

Several techniques are proposed to reduce the VMM intervention. Virtual Machine Device Queue (VMDq) [21] offloads packet classification to the network adaptor, which can directly put received packets into the guest buffer. However, it still needs VMM intervention for memory protection and address translation. Direct I/O assigns a dedicated device to each VM with I/O Memory Management Unit (IOMMU), which translates the I/O device DMA addresses to the proper physical machine addresses [3]. Direct I/O offloads VMM intervention, in memory protection and address translation, to IOMMU and allows VM to directly access I/O device, without VMM intervention for performance data movement. However, Direct I/O sacrifices device sharing and lacks of scalability, two important functionalities of virtualization, and is thus suboptimal.

Single Root I/O Virtualization (SR-IOV) [14] proposes a set of hardware enhancements for the PCIe device, which aims to remove major VMM intervention for performance data movement, such as the packet classification and address translation. SR-IOV inherits Direct I/O technology through using IOMMU to offload memory protection and address translation. An SR-IOV-capable device is able to create multiple "light-weight" instances of PCI function entities, known as Virtual Functions (VFs). Each VF can be assigned to a guest for direct access, but still shares major device resources, and thus achieve both resource sharing and high performance.

987-1-4244-5659-8/09/$25.00 ©2009 IEEE

This paper proposes a generic virtualization architecture for SR-IOV devices, which can be implemented on various kinds of VMM. It contains a VF driver, a PF driver, and an SR-IOV Manager (IOVM). The VF driver runs in the guest OS as a normal device driver, the PF driver in service OS (domain 0 in Xen) to manage Physical Function (PF), and the IOVM in VMM. The communication among them goes directly through the SR-IOV devices, so that their interface does not depend on the VMM interface. This makes the architecture independent of underlying VMM.

We carry out comprehensive experiments to evaluate SR-IOV performance and compare it with a paravirtualized NIC driver. The result shows SR-IOV virtualization can achieve 10 Gbps line rate. SR-IOV can also scale network to 60 VMs at the cost of 1.76% additional CPU overhead in a paravirtualized virtual machine (PVM) and 2.8% in a hardware virtual machine (HVM), per VM, without sacrificing throughput, while PV NIC driver suffers from excessive CPU overhead.

The results show SR-IOV has better throughout and scalability and lower CPU utilization. It shows SR-IOV provides a good virtualization solution with high I/O performance.

The rest of the paper is organized as follows: In section II, we give a more detailed introduction to SR-IOV. Section III describes the common SR-IOV architecture and some design considerations. Section IV discusses the new virtualization overhead exposed and optimizations to reduce them. Section V gives performance evaluation results and section VI discusses related work. We conclude the paper in Section VII.

II. SR-IOV INTRODUCTION

From the software point of view, a device interacts with the processor/software in three ways – namely, interrupt, register and shared memory, as shown in Figure 1. Software programs the device through registers, while the device notifies the processor through asynchronous interrupt. Shared memory is widely implemented for device to communicate with the processor for massive data movement through DMA [3].

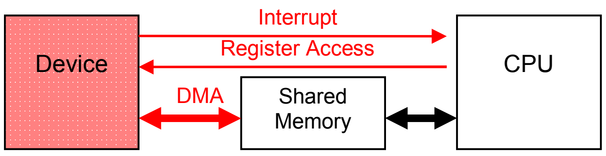

Figure 1 Device interacts with processor through interrupt, register and shared memory

SR-IOV is a new specification released by the PCI-SIG organization, which proposes a set of hardware enhancements to the PCIe device. It aims to remove major VMM intervention for performance data movement. SR-IOV inherits Direct I/O technology through using IOMMU to offload memory protection and address translation.

An SR-IOV-capable device is a PCIe device, which can be managed to create multiple VFs. A PCIe function is a primary entity in the PCIe bus, with a unique requester identifier (RID), while a PCIe device is a collection of one or more functions. An SR-IOV-capable device has single or multiple Physical Functions (PFs), as shown in Figure 2. Each PF is a standard PCIe function associated with multiple VFs. Each VF owns performance-critical resources, dedicated to a single software entity to support performance data movement in runtime, while sharing major device resources, such as network physical layer processing and packet classification, as shown in Figure 3. It is viewed as a "light-weight" PCIe function, configured and managed by PFs.

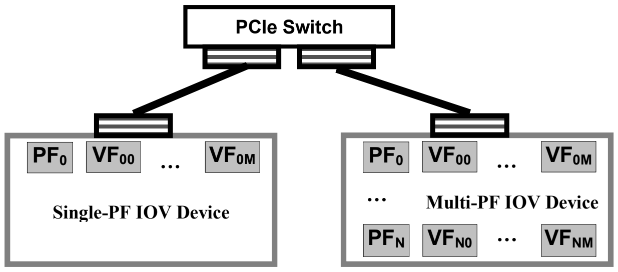

Figure 2 SR-IOV-capable devices

A VF is associated with a unique RID, which uniquely identifies a PCIe transaction source. RID is also used to index the IOMMU page table, so that different VMs can use different page tables. IOMMU page table is used for memory protection and address translation in runtime DMA transaction. Resources for device initialization and configuration, such as PCIe configuration space registers implemented in conventional PCIe devices, are no longer duplicated in each VF so that the device can have more VFs within the limited chip design budget and thus better scalability, comparing with conventional multi-function PCIe devices [14].

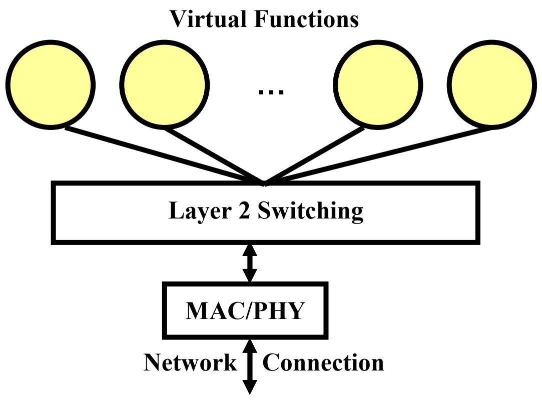

Figure 3 Resource sharing in an SR-IOV-capable network device

SR-IOV also introduces Address Translation Services [14] to provide better performance. It requires an I/O device to participate in the address translation mechanism, by caching the I/O Translation Lookaside Buffer (I/O TLB) locally, as Address Translation Cache. This enables the device to translate DMA addresses, prior to issuing the transactions, and avoid I/O TLB miss penalties during address translation in IOMMU.

III. Design of a Virtualized I/O Architecture Based on SR-IOV Devices

In this section, we propose a generic virtualization architecture for SR-IOV devices, which can be implemented on various kinds of VMM [2]. The architecture is independent of underlying VMM, allowing VF and PF drivers to be reused across different VMM, such as Xen and KVM [7]. The VF can even run in a native environment with a PF driver, within the same OS.

In this section, we also discussed a number of design considerations for the generic virtualization architecture. For example, we have special considerations for the communication between the PF and VF drivers, so that their interface does not depend on VMM interface. We also discussed the security consideration.

We implemented the generic virtualization architecture for an SR-IOV-capable network device. As the implementation of the architecture components is agnostic of underlying VMM, the implementation is ported from Xen to KVM, without code modification to the PF and VF drivers. Our implementation has been incorporated into formal releases of Xen and KVM distributions.

Our initial implementation with the network interface card (NIC) exposes new virtualization overhead, although the well-known packet classification and address translation overhead is eliminated. VF interrupt is still intervened by VMM to remap it from host to guest, which imposes major performance overhead in SR-IOV. On the critical path of interrupt handling, the most time consuming tasks are: emulation of a guest interrupt mask and unmask operation and End of Interrupt (EOI).

Three optimizations are applied to reduce virtualization overhead: interrupt mask and unmask acceleration optimization moves the emulation of interruption mask and unmask from user level application to hypervisor; and virtual EOI acceleration optimization further reduces virtualization overhead, using hardware-provided decode information. Adaptive interrupt coalescing optimization, specialized for virtualization usage model, achieves minimal CPU utilization, without sacrificing throughput.

A. Architecture

The architecture contains the VF driver, the PF driver, and the SR-IOV Manager (IOVM). The VF driver runs in a guest OS as a normal device driver, the PF driver in a service OS (host OS, or domain 0 in Xen) to manage Physical Function (PF), and IOVM runs in the service OS to manage control points within PCIe topology and presents a full configuration space for each VF. Figure 4 illustrates the architecture.

To make the architecture independent of underlying VMM, each architecture component needs to avoid using an interface specific to a VMM. For example, the communication between PF driver and VF driver goes directly through the SR-IOV devices, so that their interface does not depend on VMM interface.

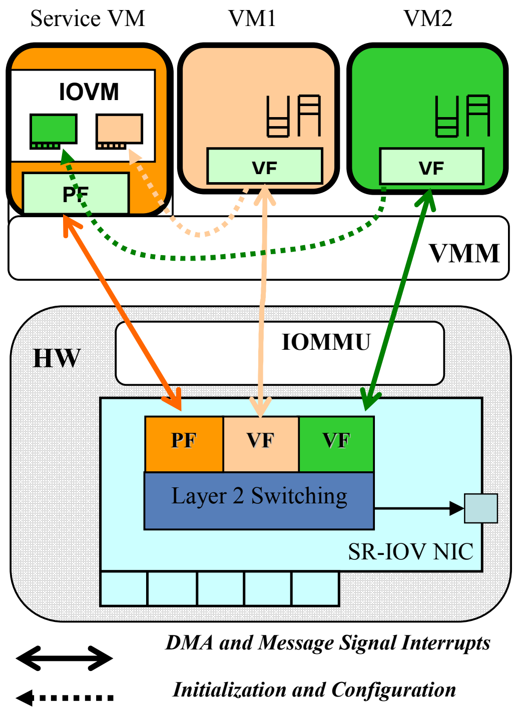

Figure 4 SR-IOV Virtualization Architecture

- PF driver

The PF driver directly accesses all PF resources and is responsible for configuring and managing VFs. It sets the number of VFs, globally enables or disables VFs, and sets up device-specific configurations, such as Media Access Control (MAC) address and virtual LAN (VLAN) settings for a network SR-IOV device. The PF driver is also responsible for configuring layer 2 switching, to make sure that incoming packets, from either the physical line or from other VFs, are properly routed. In the case of broadcast or multicast packets, the device replicates the packets to each of the physical and virtual functions, as needed.

- VF driver

The VF driver runs on the guest OS as a normal PCIe device driver and accesses its dedicated VF directly, for performance data movement, without involving VMM. From a hardware cost perspective, a VF needs only to duplicate performance critical resources, such as DMA descriptors etc., while leaving those nonperformance-critical resources emulated by IOVM and PF driver.

- IOVM

IOVM presents a virtual full configuration space for each VF, so that a guest OS can enumerate and configure the VF as an ordinary device. When a host OS initializes the

SR-IOV-capable device, it cannot enumerate all its VFs by simply scanning PCIe function vendor ID and device ID. Because a VF is a trimmed "light-weight" function, it does not contain full PCIe configuration fields and does not respond to an ordinary PCI bus scan. Our architecture uses Linux PCI hot add APIs to dynamically add VFs to the host OS. And then VFs can be assigned to the guest. The architecture only depends on a minimal extension to the existing Linux PCI subsystem to support SR-IOV implementation.

Once a VF is discovered and assigned to the guest, the guest can initialize and configure the VF as an ordinary PCIe function, because IOVM presents a virtual full configuration space for each VF. This work could be done in a user level application, such as device model in Xen for HVM, or kernel level backend service driver, such as PCIback, for PVM.

The critical path for performance in SR-IOV is to handle the interruption coming from a network device. VMM intervention to a performance packet in SR-IOV is eliminated. The layer 2 switching classifies incoming packets, based on MAC and VLAN addresses, directly stores the packets to the recipient's buffer through the DMA, and raises a Message Signal Interrupt (MSI), or its extension, MSI-x [14]. (In this paper, MSI and MSI-x are interchangeable, representing both cases, except when explicitly noted). IOMMU remaps the recipient's DMA buffer address, from the VF driver programmed guest physical address to the machine's physical address. Xen captures the interrupt and recognizes the owner guest by vector, which is globally allocated to avoid interrupt sharing [3]. It then signals a virtual MSI interrupt to the guest for notification. The guest VF driver gets executed with the virtual interrupt, and reads the packets in its local buffer. The VF driver is then able to directly operate the VF's runtime resources to receive the packets and clean up the events, such as advancing the head or tail of the descriptor ring. A single guest interrupt may handle multiple incoming packets and modern NICs, such as the Intel 82576 [6], which provides the capability to quiesce interrupt firing for a certain time to moderate the interrupt frequency.

B. PF Driver/VF Driver Communication

The PF and VF drivers need a channel of communication between them to transmit configuration and management information, as well as event notification. For example, the VF driver needs to send requests from the guest OS, such as setting up a list of multicast addresses and VLAN to the PF driver. The PF driver also needs to forward some physical network events to each VF driver, to notify the change of resource status. These events include impending global device reset, link status change, and impending driver removal etc.

In our architecture, the communications between the VF and PF drivers depends on a private hardware-based channel, which is specific to the device. The Intel SR-IOV-capable network device, the 82576 Gigabit Ethernet Controller, implemented that type of hardware-based communication method with a simple mailbox and doorbell system. The sender writes a message to the mailbox and then "rings the doorbell", which will interrupt and notify the receiver that a message is ready for consumption. The receiver consumes the message and sets a bit in a shared register, indicating acknowledgement of message reception.

C. Security Consideration

SR-IOV provides a secured environment, which allows the PF driver to monitor and enforce policies concerning VF device bandwidth usage, interrupt throttling, congestion control, broadcast/multicast storms, and etc., to enforce performance and security isolation between VMs. The PF driver inspects configuration requests from VF drivers and monitors behavior of the VF drivers and the resources they use. It may take appropriate action if it finds anything unusual. For example, it can shut down the VF assigned to a VM, if it suffers a security breach and malicious behavior.

IV. Optimization

SR-IOV aims for scalable and high performance I/O virtualization, by removing VMM intervention for performance data movement, without sacrificing device sharing across multiple VMs. However, VMM needs to intercept guest interrupt delivery, inherited from Direct I/O. It captures the physical interrupt from the VF, remaps it to the guest virtual interrupt and injects the virtual interrupt instead. The VMM also needs to emulate the guest interrupt chip, such as virtual Local APIC (LAPIC) for the HVM guest and event channel [1] for the Xen PVM guest.

The overhead of VMM intervention to interrupt delivery could be non-trivial. In a high-speed I/O device, such as a network, the interrupt frequency could be up to 70 K per second per queue in the Intel Gigabit Ethernet driver, known as the IGB driver, for lowest latency. The intervention could then be a potential performance and scalability bottleneck in SR-IOV, and thus preventing SR-IOV from reaching its full potential. We want the highest throughput, with reasonable latency specialized for virtual environment. In this section, we will discuss, in detail, our work on identifying and resolving these obstacles, which pave the way for highly scalable SR-IOV implementation.

Subsection A presents the hardware and software configuration employed in our experiment. Then, in subsection B, we introduce interrupt mask and unmask acceleration optimization, which reduces domain 0 cost from 30% to ~3% in the case of 7 VMs, followed by virtual EOI acceleration in subsection C, which can further reduce virtualization overhead by 28%. Finally, we explored adaptive interrupt coalescing in subsection D, which achieves minimal CPU utilization, without sacrificing throughput.

A. Experimental Setup

Due to the unavailability of 10 Gbps SR-IOV NIC at the time we started the research, we use ten port Gigabit SR-IOV Intel 82576 NICs as a replacement in section V. For simplicity, the optimization in this section is experimented with single port 1 Gbps network. The rest of the experiment is set up as follows, except where explicitly noticed:

The "server" system is an Intel® Xeon® 5500 platform, equipped with two quad-core processors with SMT-enabled (16 threads in total), running at 2.8 GHZ, with 12 GB 3-channel DDR3 memory. Two 4-port and one 2-port Intel 82576 Gigabit NICs are used, to provide aggregate 10 Gbps bandwidth. Each port is configured to have 7 VFs enabled, as shown in Figure 5. When 10*n VMs are employed in subsection C of section V, the assigned VFs will come from VF_{7j+0} to VF_{7j+n-1} for each port j.

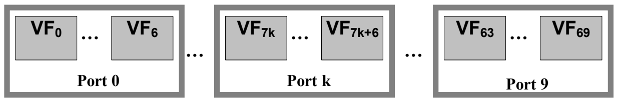

Figure 5 VF and port allocation

Xen 3.4 64 bit version hypervisor is employed in 'server' and 64 bit RHEL5U1 for both domain 0 and guest. Domain 0 employs 8 VCPUs and binds each of them to a thread in different core, and guest runs with only one VCPU which is bounded to the rest threads evenly. IGB driver 1.3.21.5 is run as a PF driver in domain 0. The guest is assigned a dedicated VF and runs VF driver version 0.9.5, with netperf benchmark running as server, to measure the receive side performance and scalability.

The "client" has the same hardware configuration as the "server", but runs 64 bit RHEL5U1 in the native. A netperf with 2 threads is run to avoid CPU competition, pairing one netperf with the "server" guest. All the NICs have support for TSO, scatter-gather I/O, and checksum offload. The "client" and "server" machines are directly connected.

B. Interrupt Mask and Unmask Acceleration

Performance of SR-IOV virtualization, before optimization, is shown in Figure 6 for HVM. The horizontal axis is labeled as VMn, where n means the number of VMs created with the same configuration. All VFs come from the same port.

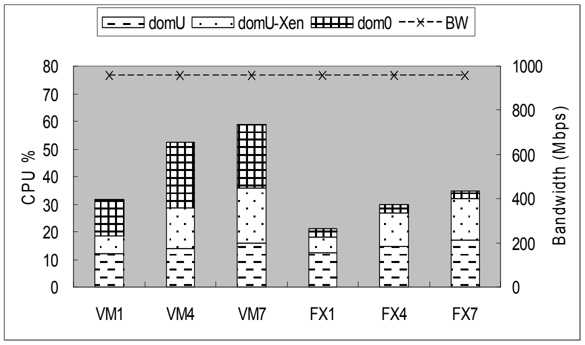

Figure 6 CPU utilization and throughput in SR-IOV with a 64 bit RHEL5U1 HVM guest

The experiment shows that SR-IOV network virtualization has almost perfect throughput, but suffers from excessive CPU utilization in domain 0. As VM# scales from 1 to 7, the throughput keeps almost exactly flat, close to the physical line rate. However, an excessive domain 0 CPU utilization is observed, starting from 17% in the case of 1 VM, to 30% in the case of 7 VMs, which could increase more as the VM# increases. Device model, or IOVM, which is an application used to emulate a virtual platform for an HVM guest, comes to the top of the CPU cycle consumers in domain 0.

Virtual MSI emulation was root caused to be the source of additional CPU utilization. Instrument code in the device model finds that the guest frequently writes MSI mask and unmask registers, which hereby trigger VM-exit [24] to hypervisor, and is forwarded to device model for emulation. This imposes not only a domain context switch between guest and domain 0, but also task context switches within domain 0 guest OS. Inspecting the RHEL5U1 guest OS source code, which is built based on Linux version 2.6.18, finds it masks interrupt at very beginning of each MSI interrupt handling and unmasks the interrupt after it completes.

By moving the emulation of interruption mask and unmask from user level application to hypervisor, domain 0 CPU utilization is significantly reduced to ~3% in all cases from 1 VM to 7 VMs. The result is shown in Figure 6 as well, with data labeled as FXn. The optimization also mitigates TLB and cache pollution caused by frequent context switching between the guest and domain 0. Both the guest and Xen CPU utilization are observed to drop slightly after optimization although the code path executed is still same.

For the edge triggered MSI interrupt, mask and unmask is not a must. Linux actually implemented the runtime mask and unmask operation only in its 1st version of MSI support kernel, which is in 2.6.18. It is removed later on, to avoid the runtime cost, since the operation itself is very expensive, even in the native environment.

C. Virtual EOI Acceleration

The OS writes an end-of-interrupt (EOI) [5] register in LAPIC to inform the hardware that interrupt handling has ended, so the LAPIC hardware can clear the highest priority interrupt in servicing. In an HVM container, based on Intel Virtualization Technology, a guest EOI write will trigger an APIC-access [5] VM-exit event, for VMM to trap and emulate.

Virtual EOI is identified as another major hot spot in SR-IOV. Tracing all VM-exit [5][24] events in Xen, to measure the CPU cycles spent, from the beginning of the VM-exit to the end, in hypervisor under same configuration demonstrated in subsection B, shows that APIC-access VM-exit is the top performance bottleneck, as shown in Figure 7. 139M cycles, or 90% of total virtualization overhead, are spent in APIC-access VM-exit for single VM case. The overhead will be multiplied as the VM# increases because each VM, with a dedicated VF, will interrupt at almost the same frequency and thus process the virtual interrupt at the same pace. Among APIC-access VM-exit, 47% of them are EOI write.

Inspecting the way Xen emulates virtual EOI, we find that Xen needs to fetch, decode, and emulate the guest instruction for virtual EOI write. Fetching guest code requires walking a multi-level guest page table and inserting translations on the host side. Decode and emulation are time-consuming, while the real work of virtual EOI write emulation, invoked by instruction emulation, is pretty light weight.

Using the hardware-provided Exit-qualification VMCS field [5] can accelerate the virtual EOI emulation. The Exit-qualification field, in APIC-access VM-exit, contains the offset of access within the APIC page, as well as indication of read or write. Upon receiving a virtual EOI, the APIC device model clears the highest priority virtual interrupt in servicing, and dispatches the next highest priority interrupt to the virtual CPU, neglecting the value the guest writes to. This provides us the possibility of using the Exit-qualification field to bypass the fetch-decode-emulation process, directly invoking the virtual APIC EOI write emulation code.

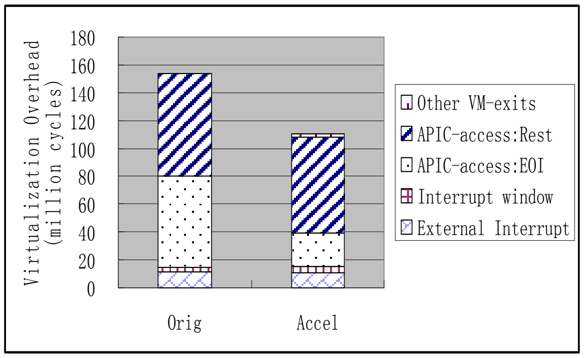

Figure 7 Virtualization overhead per second based on VM-exit events

28% of total virtualization overhead is observed to be reduced, as shown in Figure 7, after virtual EOI acceleration. The total virtualization overhead drops from the previous 154 million cycles to 111 million cycles per second.

Bypassing the fetch-decode-emulation process reduces the virtual EOI emulation cost from the original 8.4 K cycles to 2.5 K cycles each VM-exit. But this imposes an additional challenge of correctness because a guest may use *complex instruction* to write EOI and to update additional CPU states. For example, movs and stos instruction can be used to write EOI and adjust DI register, as well. Bypassing the fetch-decode-emulation process may not be able to correctly emulate the additional state transition leading to guest failure. This can be solved by checking the guest instruction, but it imposes an additional cost of 1.8 K cycles to fetch the instruction. Hardware architectures can be enhanced to provide VMM the op code to overcome the additional instruction fetching cost. On the other hand, we didn't notice any commercial OS that used this kind of *complex instruction* in practice. We argue that we do not need to worry too much because the risk is contained within the guest. We believe it is worth it to use virtual EOI acceleration, bypassing the fetch-decode-emulation process.

D. Adaptive Interrupt Coalescing

High frequency interrupts can cause serious performance problems in modern OSes, particularly for those with high bandwidth I/O devices. Frequent context switching, caused by interrupts, not only consumes CPU cycles, but also leads to cache and TLB pollution [17]. Technologies to moderate interrupts exist, such as NAPI [20], hybrid mode of interrupt disabling and enabling (DE) [19] and interrupt coalescing in modern NIC drivers, such as the IGB driver, which throttles interrupts after a certain amount of time [6][11].

SR-IOV brings additional challenges to interrupt coalescing. The coalescing policy in the driver is originally optimized for the native environment, without considering the virtualization overhead, and thus could be suboptimal. A higher than physical line rate bandwidth may be achieved in inter-VM communication, leading to much higher frequency interrupts than normal and thus may confuse the device driver. For example, the packets per seconds (pps), sampled by the device driver and interrupt throttling based on pps, may exceed expectation, and, thus, is suboptimal.

Reducing interrupt frequency can minimize virtualization overhead, but it may increase network latency and thus hurt TCP throughput [28]. Longer interrupt interval also means more packets will be received during each interrupt, which could lead to packet drop, if it exceeds the internal buffer number used in the device driver, socket layer, and application. Optimal interrupt coalescing policy is particularly important to balance the tradeoff.

We experimented with an adaptive interrupt coalescing (AIC) optimization, specialized for a virtualization usage model based on overflow avoidance, to configure interrupt frequency low enough, but won't overflow the socket or application buffer, as well as the device driver buffer. We used (3):

$$bufs = \min(ap\_bufs, dd\_bufs) \tag{1}$$

$$t_d * r = bufs \;/\; pps \tag{2}$$

$$IF = 1/t_d = \max(pps/(bufs * r), lif) \tag{3}$$

Here ap_bufs and dd_bufs means the number of buffers used by the application and the device driver, while t_d and pps means the interval between two interrupts and the number of received packets per second respectively. Considering the VMM intervention introduced latency, a redundant rate, r, is used to provide time budget for hypervisor to intervene, and we limit the minimal interrupt frequency, using IF, to lif indicating the lowest acceptable interrupt frequency to limit the worst latency.

We have prototyped AIC to study the impact of interrupt coalescing in SR-IOV. Default guest configuration is employed, i.e. 64 ap_bufs (120832 B socket buffer size in RHEL5U1) and 1024 dd_bufs. An approximately 20% hypervisor intervention overhead is estimated, i.e. r =1.2, and pps is sampled per second, to adaptively adjust IF.

Figure 8 and Figure 9 show the results of bandwidth and CPU utilization vs. different interrupt coalescing policies for both UDP_STREAM and TCP_STREAM: 20 KHZ, 2 KHZ, AIC and 1 KHZ, running 64 bit Linux 2.6.28 in HVM container. 2 KHZ interrupt frequency is the VF driver's default configuration, and 20 KHZ interrupt frequency denotes the normal case used for low latency in modern NIC drivers, such as the IGB driver. CPU utilization of domain 0 remains as low as ~1.5%, among all configurations, and the throughput stays at 957 Mbps for the UDP stream and 940 Mbps for the TCP stream in the cases of 20 KHZ, 2 KHZ, and AIC. But a 9.6% drop in throughput is observed for the TCP stream, at 1 KHZ, reflecting the fact that TCP throughput is more latency sensitive. The result shows a 40% and 50% CPU utilization savings from the 20 KHZ case, to the 2 KHZ case, for UDP and TCP respectively, and can further reduce the CPU utilization in AIC.

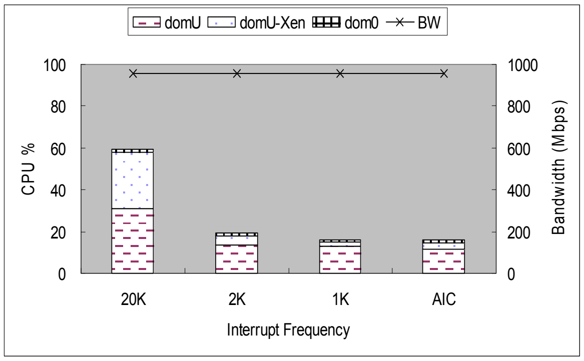

Figure 8 Adaptive interrupt coalescing reduces CPU overhead for UDP_STREAM

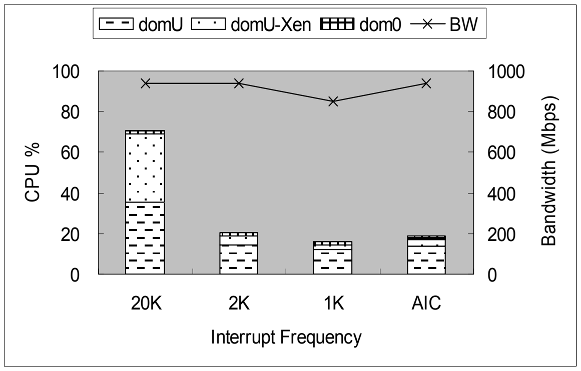

Figure 9 Adaptive interrupt coalescing maintains throughput with minimal CPU utilization for TCP_STREAM

In the meantime, AIC improves inter-VM communication performance, by avoiding packet loss, caused by insufficient receiving buffers. Figure 10 shows the throughput and CPU utilization for the case where domain 0 sends packets to guest. The transmit side bandwidth (TX BW) stays flat, while the receiving side bandwidth (RX BW) may be lower than the transmitting side. This reflects the fact that some packets are dropped due to insufficient receiving buffers, in cases of 2 K and 1 K interrupt frequency cases. The interrupt frequency in AIC increases adaptively as the throughput increases, to avoid packet loss, while fixed interrupt frequency suffers from either excessive CPU utilization, in the case of 20 KHZ case, or throughput drop in, the cases of 2 KHZ and 1 KHZ.

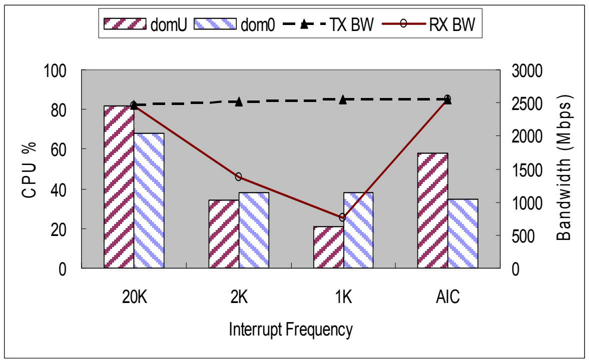

Figure 10 Adaptive interrupt coalescing avoids packets lose in inter-VM communication

In summary, AIC achieves minimal CPU utilization, without sacrificing throughput, so it is a good tradeoff.

V. EXPERIMENT

Incorporating the above 3 optimizations, this section evaluates the performance and scalability of the generic virtualization architecture for SR-IOV devices with aggregate 10 Gbps configuration, except subsection B which uses the single port environment. 64 bit Linux 2.6.28 is used in the HVM container to take advantage of tickless idle, except where noted.

A. SR-IOV Network Performance

Figure 11 shows the result of CPU utilization reduction with MSI, EOI, and AIC optimizations for both Linux 2.6.18 HVM and Linux 2.6.28 HVM. SR-IOV achieves a 10 Gbps line rate in all situations, but CPU utilization is reduced dramatically with our optimizations. MSI optimization reduces CPU utilization from the original 499% to 227% for a Linux 2.6.18 HVM guest, which frequently accesses MSI mask and unmask registers at runtime. The majority (208%) of savings comes from domain 0, but the guest also contributes 16% and Xen contributes an additional 48%, as a result of TLB and cache pollution mitigation. A Linux 2.6.28 HVM guest, which does not access MSI mask and unmask registers at runtime, is observed to have 23% CPU utilization reduction with EOI optimization, and a further 24% reduction with AIC optimization. With all optimizations, SR-IOV can achieve 9.57 Gbps with 193% CPU utilization, which is only 48%

higher than in the native, where 10 VF drivers run in the same OS, with PF drivers on top of bare metal.

B. *Inter-VM Communication Performance*

Packets of inter-VM communication in SR-IOV are internally switched in NIC, without going through the physical line, and thus will be able to exceed the full line rate. Up to 2.8 Gbps throughput is achieved in SR-IOV with a single port, as shown in Figure 12. The PV counterpart achieves 4.3 Gbps with more CPU utilization, as shown in Figure 13.

The inter-VM communication throughput is subjected to NIC hardware implementation in SR-IOV. The device uses DMA to copy packets from source VM memory to NIC FIFO, and then from NIC FIFO to target memory. Both DMA operations need to go through slow PCIe bus transactions, which limit the total throughput. In paravirtualization, the packets are directly copied from source VM memory to target VM memory by CPU, which operates on system memory in faster speed and thus achieves higher throughput than SR-IOV. Further more, as the message size goes up from 1500 bytes to 4000 bytes, each system call consumes more data and thus less overhead spent in the network stack, leading to higher bandwidth as well. However, in terms of throughput per CPU utilization, SR-IOV is better.

C. *Scalability*

SR-IOV has almost perfect I/O scalability for both HVM and PVM, as shown in Figure 14 and Figure 15. It can keep up to the physical line rate (9.57 Gbps) consistently, with very limited additional CPU cycles, from 10 VMs to 60 VMs. HVM spends an additional 2.8% CPU cycles on each additional guest, while PVM spends 1.76%. This is because Xen PVM implements a paravirtualized interrupt controller, known as event channel [1], which consumes fewer CPU cycles than virtual LAPIC in HVM. An interesting finding is that, in our experiment, PVM consumed slightly more CPU cycles than HVM, in the case of 10 VMs . This is because the user and kernel boundary crossing, in guest X86-64 XenLinux, needs to go through the hypervisor to switch the page table for isolation [12].

D. *Comparing with PV NIC*

Scalability of PV NIC suffers from excessive CPU utilization leads by packets copy in domain 0. The existing Xen PV NIC driver uses only a single thread in the backend to copy packets, which can easily saturate at 100% CPU utilization. So using that driver does not scale well with additional CPU cores. It can achieve only 3.6 Gbps in our experiment in the case of 10 VMs, and the throughput drops as the VM# keeps increasing. We enhanced the Xen PV NIC driver to accommodate more threads for backend service, so that it could take the advantage of multi-core CPU computing capability, for fair comparison.

The scalability results of enhanced PV NIC driver are shown in Figure 16 and Figure 17 for HVM and PVM respectively. The CPU utilization goes up and the throughput drops when the VM number increases in both cases. The domain 0 CPU utilization in HVM case is slightly higher than that in PVM case (431% vs. 324%), this is because of the additional layer of interrupt conversion between event channel and conventional LAPIC interrupt. For PV NIC in HVM guest, the event channel mechanism, originated in PVM, is built on top of conventional LAPIC interrupt mechanism. But the guest in PVM case consumed slightly more CPU cycles than that in HVM case due to the hypervisor involvement and page table switch for each system call in X86-64 XenLinux, same as that in subsection C.

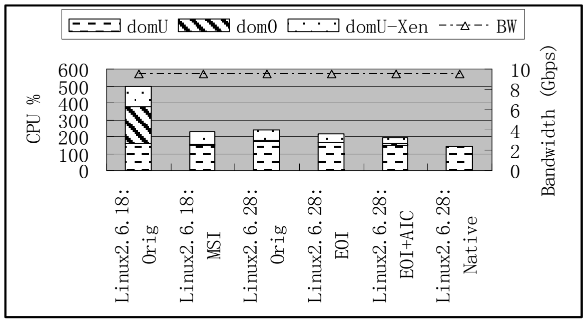

Figure 11 Impact of the optimizations for SR-IOV with aggregate 10 Gbps Ethernet

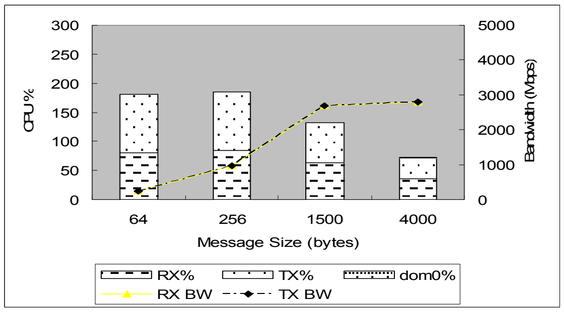

Figure 12 SR-IOV inter-VM communication

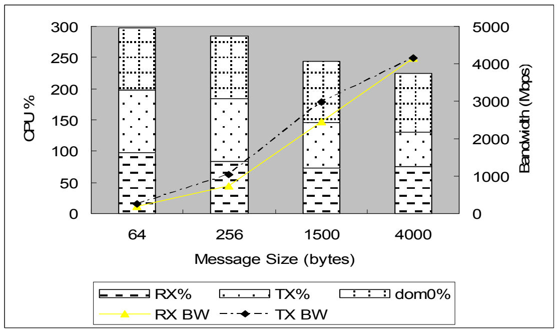

Figure 13 PV NIC inter-VM communication

VMDq [21] is another adjacent technology which is closely related to SR-IOV. However, as to the day of this research work, we are unable to get an stable tree to repeat what [16] did and thus unable to compare between VMDq and SR-IOV fairly.

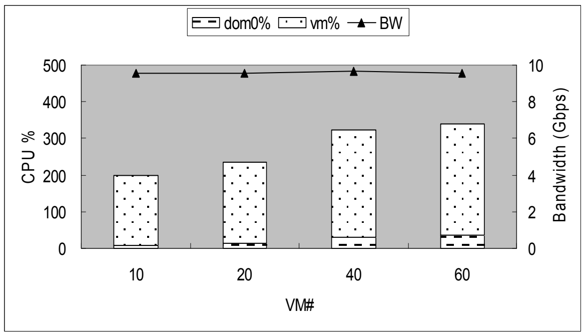

Figure 14 SR-IOV scalability in HVM

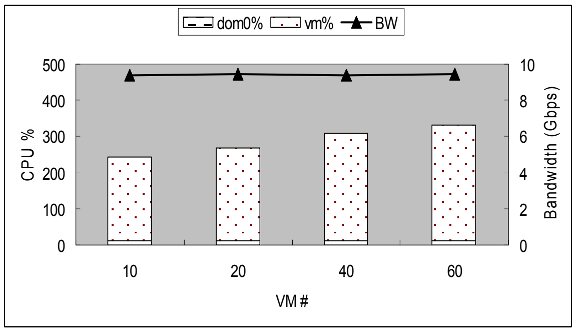

Figure 15 SR-IOV scalability in PVM

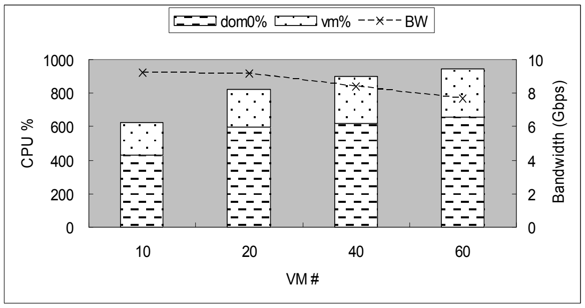

Figure 16 PV NIC scalability in HVM

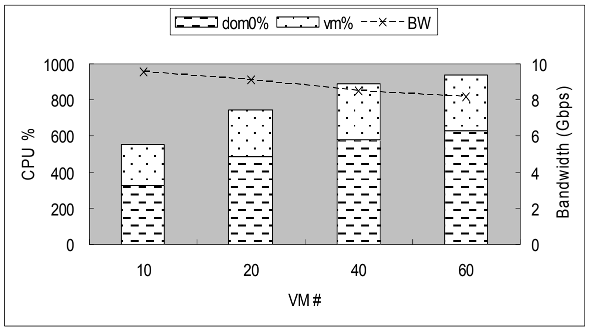

Figure 17 PV NIC scalability in PVM

VI. RELATED WORK

Researchers from both academia and industry areas have done a lot of exploration of I/O virtualization. Virtualization and performance of Ethernet adapter on Vmware Workstation [23][22] have been studied. Xen's PV driver [4] model uses a shared-memory-based data channel and customized interface to reduce data movement overhead, between the guest and domain 0. Page remapping and batch packets transferring could be used to further improve performance [10]. Cache-aware scheduler, virtual switch enhancement and transmit queue length optimization are studied [8]. They all suffer from the overhead of extra data copy. Xenloop [25] improves inter-VM communication performance using shared memory, but requires new user APIs. I/O latency caused by VM scheduling is mitigated with a variety of scheduler extensions [13], but cannot be fully eliminated.

Various hardware-assisted solutions are proposed in virtualization to achieve high-performance I/O, such as VMDq [16][21] and self-virtualized devices [15]. In these solutions, each VM is assigned portion of resource, a pair of queues, to transmit and receive packets, but VMM involvement is still required in packet processing, for memory protection and address translation. Optimization, such as grant-reuse, can be used to mitigate virtualization overhead. Neither of the above technologies can fully eliminate VMM involvement for performance data movement, so they suffer from resulting performance issues, as well.

IOMMU was recently applied to commercial PC-based systems to offload memory protection and address translation, attracting a lot of new attention. Willmann studied possible protection strategies with IOMMU [26]. Dong implemented Direct I/O in Xen, based on IOMMU, for a variety of PC devices [3]. Direct I/O achieves close to native performance, but suffers from sharing and security issues. VMM-Bypass I/O is explored [9], extending OS-bypass design of InfiniBand software stack to bypass VMM for performance as well. But it cannot be extended to the majority of devices that do not implement OS-bypass software stack such as the Ethernet, while SR-IOV implements in generic PCIe layer which is already adopted by modern commercial OSs.

Interrupt is identified as one of the major performance bottlenecks for high bandwidth I/O, due to frequent context switching, as well as potential cache and TLB pollution. Mechanism with combination of polling and interrupt disabling/enabling is explored in the native OS, to reduce this overhead. For example polling-based NAPI [20] in Linux, interrupt disabling and enabling (DE), and hybrid mode [19] are explored to mitigate the interrupt overhead. Neither of them explores interrupt coalescing in a virtual environment.

VII. CONCLUSION

We designed, implemented, and tuned a generic virtualization architecture for an SR-IOV-capable network device, which supports reusability of PF and VF drivers across different VMMs. On the critical path of interrupt handling, the most time consuming tasks are emulation of guest interrupt mask and unmask operation and End of Interrupt (EOI). Three optimizations are applied to reduce virtualization overhead: Interrupt mask and unmask acceleration reduce CPU utilization from the original 499% to 227%; Virtual EOI acceleration and adaptive interrupt coalescing further reduce CPU cycles, by 23% and 24% respectively. Based on our implementation, we conducted performance measurement to compare SR-IOV solution with others. Our experimental data proved that SR-IOV provides a good solution for a secure and high performance I/O virtualization.

As CPU computing capability continues to scale with Moore's Law, along with multi-core technology, it's expected to run more VMs on a single platform. SR-IOV-based I/O virtualization serves as a good base to meet the scalability requirement. We believe that SR-IOV-based I/O virtualization provides a good platform for further study of I/O virtualization such as video and audio sharing.

Acknowledgment

The other members contributing to this study include Qing He, Dongxiao Xu, Ronghui Duan and Intel Linux OS VMM team. We would like to thank Richard Uhlig, Nakajima Jun, Wilfred Yu for their excellent insights and suggestions. We also would like to thank the helpful academic works sponsored by China Basic Research 973 program (Grant No. 2006CB303003) and the National Natural Science Foundation of China (Grant No. 60773093, 60970107, 60970108).

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, Xen and the art of virtualization, In proceedings of the 19th ACM symposium on Operating Systems Principles, Bolton Landing, NY, 2003, 164-177

[2] Y. Dong, et al. SR-IOV Networking in Xen: Architecture, Design and Implementation. 1st Workshop on I/O Virtualization, San Diego, CA, 2008

[3] Y. Dong, J. Dai, et al. Towards high-quality I/O virtualization. Proceeding of the Israeli Experimental Systems Conference (SYSTOR), Haifa, Israel 2009

[4] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams, Safe hardware access with the Xen virtual machine monitor, In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, Boston, MA, 2004.

[5] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developers' Manual, http://www.intel.com/ products/processor/manuals/index.htm.

[6] Intel Corporation, Intel® 82576 Gigabit Ethernet Controller Datasheet. http://www.intel.com

[7] Kernel Virtual Machine, http://kvm.qumranet.com/

[8] G. Liao, D. Guo, L. Bhuyan, S. R King, Software techniques to improve virtualized I/O performance on multi-core systems. Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, San Jose, CA, 2008, 161-170

[9] J. Liu, et al. High Performance VMM-Bypass I/O in Virtual Machines. Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2006, 3-3

[10] A. Menon, A. L. Cox, W. Zwaenepoel, Optimizing Network Virtualization in Xen. Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2006, 15-28.

[11] J. C. Mogul, K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. ACM Transactions on Computer Systems (TOCS). 15(3), New York, NY, ACM, 1997, 217 – 252.

[12] J. Nakajima, A. Mallick, X86-64 XenLinux: Architecture, Implementation, and Optimizations, Proceeding of Ottawa Linux Symposium (OLS), 2006.

[13] D. Ongaro, A. L. Cox and S. Rixner, Scheduling I/O in Virtual Machine Monitors. Proceedings of 4th ACM/USENIX International Conference on Virtual Execution Environments, Seattle, WA, 2008, 1-10.

[14] PCI Special Interest Group, http://www.pcisig.com/home

[15] H. Raj, K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices. Proceedings of the 16th international symposium on high performance distributed computing, Monterrey, CA, 2007

[16] 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, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Washington, DC, 2009.

[17] K. Ramakrishnan, Performance consideration in designing network interfaces, IEEE Journal on Selected Areas in Communications 11(2), 1993 203–219.

[18] M. Rosenblum, T. Garfinkel. Virtual Machine Monitors: Current technology and future trends. Computer, 38(5), Los Alamitos, CA, IEEE Computer Society Press, 2005, 39-47.

[19] K. Salah, A. Qahtan, Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme, Computer Communications, 32(1), Newton, MA, Butterworth-Heinemann, 2009, 179-188.

[20] J. Salim, When NAPI comes to town, Proceedings of Linux 2005 Conference, Swansea, UK, August 2005.

[21] J. R. Santos, Y. Turner, G. Janakiraman and I. Pratt, Bridging the Gap between Software and Hardware Techniques for I/O Virtualization, Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2008. 29-42

[22] M. Steil. Inside VMware. 2006. http://events.ccc.de/congress/2006/Fahrplan/attachments/1132-InsideVMware.pdf

[23] J. Sugerman, G. Venkitachalam, B. Lim, Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor, Proceedings of the General Track: 2002 USENIX Annual Technical Conference, Boston, MA, 2001, 1-14

[24] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, L. Smith, Intel Virtualization Technology, 38(5), Los Alamitos, CA, IEEE Computer Society Press , 2005, 48–56

[25] J. Wang, K. Wright, and K. Gopalan, XenLoop: A Transparent High Performance Inter-VM Network Loopback, in Proceedings of the 17th international symposium on High performance distributed computing, Boston, MA, 2008, 109-118

[26] P. Willmann, S. Rixner, and A. L. Cox, Protection Strategies for Direct Access to Virtualized I/O Devices, Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2008, 15-28

[27] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

[28] M. Zec, M. Mikuc, M. Zagar, Integrated performance evaluating criterion for selecting between interrupt coalescing and normal interruption, International Journal of High Performance Computing and Networking, 3(5/6), Geneva, SWITZERLAND, Inderscience Publishers, 2005, 434-445.