J. Parallel Distrib. Comput. 72 (2012) 1471–1480

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

High performance network virtualization with SR-IOV

Yaozu Dong [a], Xiaowei Yang [a], Jianhui Li [a], Guangdeng Liao [a], Kun Tian [a], Haibing Guan [b,*]

[a] *Intel Asia-Pacific Research and Development Ltd., China*

[b] *Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China*

ARTICLE INFO

Article history:
Received 17 July 2011
Received in revised form
1 December 2011
Accepted 29 January 2012
Available online 15 February 2012

Keywords:
Virtualization
Virtual Machine
SR-IOV
Xen
Performance

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 performance. We propose a generic virtualization architecture for SR-IOV-capable devices, which can be implemented on multiple Virtual Machine Monitors (VMMs). With the support of our architecture, the SR-IOV-capable device driver is highly portable and agnostic of the underlying VMM. Because the Virtual Function (VF) driver with SR-IOV architecture sticks to hardware and poses a challenge to VM migration, we also propose a dynamic network interface switching (DNIS) scheme to address the migration challenge. Based on our first implementation of the network device driver, we deployed several optimizations to reduce virtualization overhead. Then, we conducted comprehensive experiments to evaluate SR-IOV performance. The results show that SR-IOV can achieve a line rate throughput (9.48 Gbps) and scale network up to 60 VMs, at the cost of only 1.76% additional CPU overhead per VM, without sacrificing throughput and migration.

1. Introduction

I/O performance is critical to high performance computer systems. With the rapid development of multi-core technology, computing capabilities keep increasing according to Moore's Law, but I/O performance is still suffering from both long latency PCI Express (PCIe) [21] and hardware scalability limitations, such as the limited number of PCIe slots. I/O intensive servers may waste CPU cycles, waiting for I/O data or spinning on idle cycles. This reduces system performance and sacrifices scalability.

Virtualization allows multiple OSs to share a single physical interface, maximizing the use of computer system resources. An additional software layer, called Virtual Machine Monitor (VMM) or hypervisor [25], is introduced to provide the abstraction of Virtual Machines (VMs), on top of which, each OS assumes owning resources exclusively. There are two approaches to enable virtualization. Paravirtualization (PV) [35] requires OS modification to work cooperatively with VMM. Full virtualization requires no modification, using hardware supports like Intel® VT [31]. Xen [1] is an open source VMM, which supports both paravirtualization and full virtualization. It runs a service OS in a privileged domain (domain 0) and multiple guest OSs in the guest domain.

Virtualization poses great challenges on I/O performance and its scalability. When a guest accesses the I/O device, VMM needs to intervene in the data processing to share the physical device. The VMM intervention leads to additional I/O overhead for a guest OS. Existing solutions, such as the Xen split device driver [8], also known as the PV driver, suffer from VMM intervention overhead, due to packet copy [28,23,13]. The virtualization overhead could saturate the CPU in high speed networks, such as 10 Gigabit Ethernet, impairing overall system performance. Several techniques are proposed to reduce VMM intervention. The Virtual Machine Device Queue (VMDq) [28] offloads packet classification to the network adapter and can put received packets directly 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 guest VM with I/O Memory Management Unit (IOMMU) translating DMA addresses from the guest programmed physical addresses to machine's physical addresses [6]. Direct I/O allows VM to directly access an I/O device, without VMM intervention. However, Direct I/O sacrifices device sharing and lacks scalability, two important capabilities of virtualization.

Single Root I/O Virtualization (SR-IOV) [21] 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, 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

* Corresponding author.
E-mail addresses: eddie.dong@intel.com (Y. Dong), xiaowei.yang@intel.com (X. Yang), jian.hui.li@intel.com (J. Li), guangdeng.liao@intel.com (G. Liao), kun.tian@intel.com (K. Tian), hbguan@sjtu.edu.cn (H. Guan).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.01.020

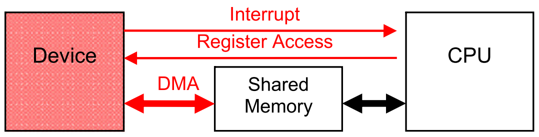

Fig. 1. The device interacts with processor through interrupt, register and shared memory.

as Virtual Functions (VFs). Each VF can be assigned to a guest for direct access, but still shares major device resources, achieving both resource sharing and high performance.

This paper proposes a software solution to efficiently use the SR-IOV-capable devices. The contributions of this paper are:

1. This paper designs and implements a generic virtualization architecture for SR-IOV-capable devices (SR-IOV virtualization, or SR-IOV architecture for short), which can be implemented on various kinds of VMM. It contains a VF driver, a Physical Function (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 the PF, and the IOVM in VMM.
2. Three optimizations are applied to eliminate the virtualization overhead remaining in the SR-IOV virtualization. The remaining overhead includes: interrupt mask and unmask acceleration, virtual End Of Interrupt (EOI) acceleration, and adaptive interrupt coalescing.
3. A dynamic network interface switching (DNIS) scheme is proposed to address an additional challenge to virtual machine migration, introduced by the additional hardware stickiness between the VF driver and the SR-IOV-capable device. DNIS takes advantage of the OS bonding network driver mechanism and virtual hot plug support of the VF device, to switch the network interface between VF and software emulated NIC, for both performance and migration support.
4. Comprehensive experiments are conducted to evaluate the SR-IOV virtualization performance and scalability up to 60 VMs, including both paravirtualized virtual machine (PVM) and hardware virtual machine (HVM).

The results show that our new SR-IOV virtualization can achieve a 10 Gbps line rate and scale to 60 VMs, at the cost of 1.76% additional CPU overhead in a PVM and 2.8% in an HVM, without sacrificing throughput and VM migration. All of these results reveal that the new SR-IOV virtualization is a promising solution to high performance virtualization.

The rest of the paper is organized as follows: In Section 2, we give a more detailed introduction to SR-IOV, followed by the related work in Section 3. Section 4 describes the common SR-IOV architecture and some design considerations. Section 5 discusses the newly exposed virtualization overhead and optimizations to reduce them. Section 6 gives performance evaluation results. We conclude the paper in Section 7.

2. SR-IOV introduction

From the software point of view, a device interacts with the processor/software in three ways: interrupt, register, and shared memory, as shown in Fig. 1. Software programs the device through registers, while the device notifies the processor through asynchronous interrupt. Shared memory is widely implemented for the device to communicate with the processor for massive data movement through DMA [6].

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 by using IOMMU to offload memory protection and address translation.

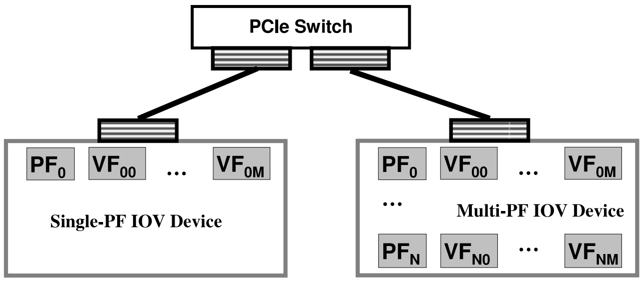

Fig. 2. SR-IOV-capable devices.

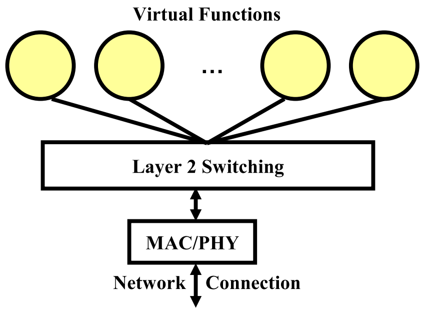

Fig. 3. Resource sharing in an SR-IOV-capable network device.

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 Fig. 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 Fig. 3. It is viewed as a "light-weight" PCIe function, configured and managed by PFs.

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. The IOMMU page table is used for memory protection and address translation in runtime DMA transactions. 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 the device can have more VFs within the limited chip design budget, resulting in better scalability, compared with conventional multi-function PCIe devices [21].

3. Related work

A wide spectrum of studies have been conducted on both native and virtualization I/O, to understand and improve performance. Authors in [14,12,15] did extensive studies on native I/O processing and proposed a new I/O architecture to improve its performance. In the area of virtualization, network performance of the Vmware Workstation [30,29] has been studied. Xen [8] uses a shared-memory-based data channel to reduce data movement overhead. Page remapping and batch packets transferring could be used to further improve performance [17]. Areas of study include: cache-aware scheduler, virtual switch enhancement, and transmit queue length optimization [13]. They all suffer from the overhead of extra data copy. Xenloop [33] 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 [20], but cannot be fully eliminated.

Various hardware-assisted solutions are proposed in virtualization to achieve high-performance I/O, such as VMDq [23,28], VMM-Bypass I/O base on InfiniBand network [16], and self-virtualized devices [22]. In these solutions, each VM is assigned a portion of the resource, such as a pair of queues, to transmit and receive packets, but VMM involvement is still required for packet processing, memory protection, and address translation. Optimization, such as grant reuse, can 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, without sacrificing sharing. Willmann studied possible protection strategies with IOMMU [34]. Dong implemented Direct I/O in Xen, based on IOMMU, on a variety of PC devices [6]. Direct I/O achieves close to native performance, but suffers from sharing and migration issues. Single Root I/O Virtualization (SR-IOV) was proposed by PCI-SIG to implement multiple "light-weight" instances of PCI function entities in devices for efficient virtualization, while sharing the majority of runtime hardware resources [21]. They require a new I/O virtualization architecture to efficiently take advantage of hardware features and migration support.

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. A mechanism with a combination of polling and interrupt disabling/enabling is explored in the native OS, to reduce this overhead. For example, polling-based NAPI [27] in Linux**, interrupt disabling and enabling (DE), and hybrid mode [26] are explored to mitigate the interrupt overhead. Neither of them explores interrupt coalescing in a virtual environment.

There is a lot of research in the literature on VM migration base on the software emulated device. Clark etc. studied live VM migration base on Xen paravirtualized devices [2]. Voorsluys etc. studied the cost of virtual machine migration in Clouds, which cannot be disregarded in systems where service availability and responsiveness are governed by strict service level agreements [32]. Zhai experimented with the Linux bonding driver for the migration of Direct I/O [37]. However, none of them can achieve 10 Gbps line rate performance and scalability without sacrificing device sharing or migration.

4. Design of a virtualized I/O architecture based on SR-IOV-capable devices

In this section, we propose a generic virtualization architecture for SR-IOV-capable devices, which can be implemented on various kinds of VMM. The architecture is independent of underlying VMM, allowing Virtual Function (VF) and Physical Function (PF) drivers to be reused across different VMM, such as Xen and KVM [11]. The VF can even run in a native environment with a PF driver, within the same OS. On the other hand, the VF driver, with SR-IOV architecture, sticks to hardware network adapters, posing an additional challenge to migration. To address the challenge, we propose a dynamic network interface switching (DNIS) scheme, which takes advantage of the OS bonding network driver and virtual hot plug support of the VF device, to switch the network interface between VF and software emulated NIC, while maintaining network connectivity.

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.

A previous version of this paper was published [5]. However, it does not address the migration issue.

4.1. 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 PF, and IOVM runs in the service OS to manage control points within PCIe topology, presenting a full configuration space for each VF. Fig. 4 illustrates the architecture.

To make the architecture independent of the 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-capable devices, so that their interface does not depend on the VMM interface.

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-capable 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.

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 presents a virtual full configuration space for each VF, so that a guest OS can enumerate and configure the VF as an ordinary PCIe 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. Then VFs can be assigned to the guest. The architecture depends only 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 a device model in Xen for a hardware virtual machine (HVM), or a backend driver, such as PCIback [8], for a paravirtualized virtual machine (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 [21]. (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 guest which owns the interrupt by vector, which is globally allocated to avoid interrupt sharing [6]. It then signals a virtual MSI interrupt to the guest for notification. The guest VF driver is 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

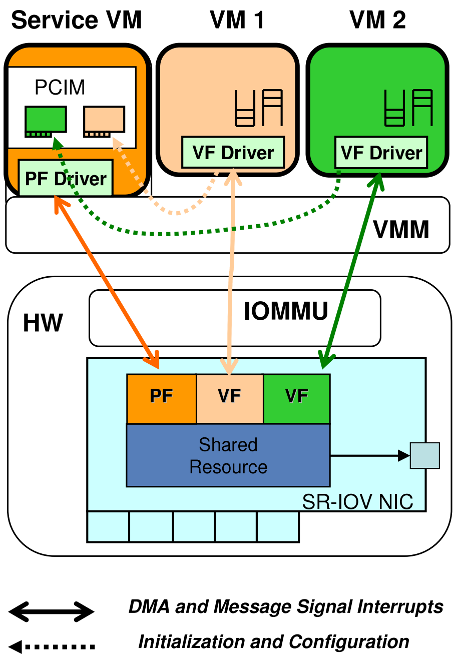

Fig. 4. SR-IOV virtualization architecture.

the head or tail of the descriptor ring. A single guest interrupt may handle multiple incoming packets in modern NICs, such as the Intel® 82 576 [10], which provide the capability to moderate the interrupt frequency.

4.2. 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 82 576 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 acknowledgment of message reception.

4.3. 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, 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.

IOVM manages control points within PCIe topology, through Access Control Service (ACS), to properly close a security hole in SR-IOV. For peer to peer (P2P) access between VFs under the same PCIe switch, it may directly route the P2P transaction, from a source downstream port to a target downstream port, without traversing upstream, bypassing IOMMU. This exposes a security issue, where a VF assigned to guest can access other VFs’ memory-mapped I/O (MMIO) areas through P2P transaction. ACS provides the capability of upstream forwarding control to close the security hole. ACS defines a set of control points within PCI Express topology, to determine whether a Transaction Layer Packet should be routed normally, blocked, or redirected to upstream. When redirection of a downstream port is turned on, the P2P request, going through the downstream port, is not routed directly within the switch. Instead, the request is forwarded to the upstream port of the switch and finally reaches Root Complex and IOMMU, for memory protection and address translation.

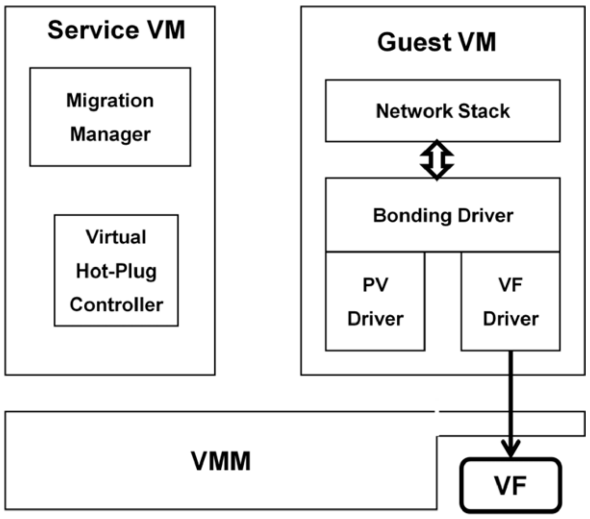

Fig. 5. VF migration with DNIS.

4.4. Migration

VM migration, that is migrating an entire VM from a source platform to a target platform, is one of the key benefits introduced by virtualization [2,3]. It enables dynamic workload replacement without any modification to the application or OS. However, network virtualization with SR-IOV imposes additional challenges on migration: (1) the target platform may not have same VF hardware or SR-IOV-capable devices as the source platform, which the VF driver assumes to run on (that is, hardware stickiness), and (2) the state of VF hardware is unlikely to be cloned easily, because the design of VF specification inherits a great deal from the conventional network interface card, which is designed for a native usage model, not migration support.

To address the migration challenge, we propose a dynamic network interface switching (DNIS) scheme, which takes advantage of the OS bonding network driver and virtual hot plug support of the VF device. An OS bonding driver, such as a Linux bonding driver [4], aggregates multiple underlying network interface drivers, and presents the OS network stack as a single logical network interface driver. The OS bonding driver chooses one network interface driver to be activated, while leaving the rest to standby. DNIS aggregates the VF driver with a software emulated virtual NIC driver, such as a PV NIC driver, for both performance and migration. It activates the VF driver at run time for performance, but switches to PV NIC driver at migration time, to eliminate hardware stickiness during migration. DNIS architecture is shown in Fig. 5.

The DNIS achieves migration through virtual hot removal of a VF device. When the migration starts, first the migration manager tells the virtual hot-plug controller to signal a virtual hot removal event of the VF device to the guest VM. The bonding driver then switches the active network driver from the VF to the PV NIC, which is hardware neutral and can be migrated seamlessly. Then, the guest OS shuts down the VF driver instance, in response to the hot removal event, to eliminate hardware stickiness, and relies on the PV NIC to maintain network connectivity. Finally, after completing the virtual hot removal event, the migration manager starts the "real" VM migration process, as if the guest was never equipped with the VF hardware. DNIS achieves VM migration without sacrificing network connectivity.

DNIS maintains network performance across migration, by adding back the VF device through virtual hot add-on at target platform if the VM has dedicated VF resource(s). The migration manager may generate a virtual hot add-on event at target platform for the presence of the VF hardware, so that the guest OS may create the VF driver instance again for performance at runtime, after migration. An additional advantage of mobile pass through is that the VF hardware in the target platform may or may not be identical to that in the source platform. We extended Xen to implement the virtual ACPI hot-plug controller device model to support the virtual hot-plug event, and adopted the Linux bonding driver to support migration of network virtualization with SR-IOV.

5. Optimizations

SR-IOV aims for high performance I/O virtualization, by removing VMM intervention and 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 Virtual Function (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 hardware virtual machine (HVM) guests and event channel [1] for Xen paravirtualized virtual machine (PVM) guests.

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 70,000 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, preventing SR-IOV from reaching its full potential. We want the highest throughput, with reasonable latency specialized for a virtual environment. In this section, we will discuss our work identifying and resolving these obstacles, which paves the way for highly scalable SR-IOV implementation (refer to Section 6 for the experiment set up).

We introduce interrupt mask and unmask acceleration optimization in Section 5.1 to reduce domain 0 cost, followed by virtual EOI acceleration in Section 5.2. In Section 5.3, we explore adaptive interrupt coalescing to achieve minimal CPU utilization.

5.1. Interrupt mask and unmask acceleration

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

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 remains 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.

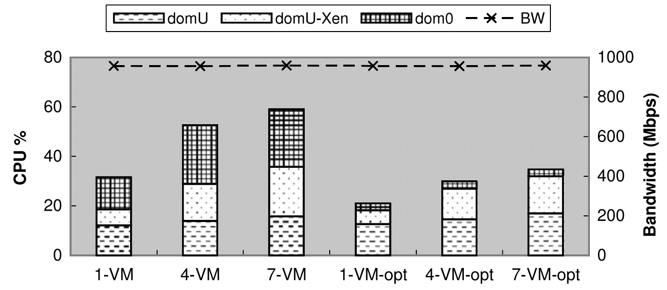

Fig. 6. CPU utilization and throughput in SR-IOV with a 64 bit RHEL5U1 HVM guest.

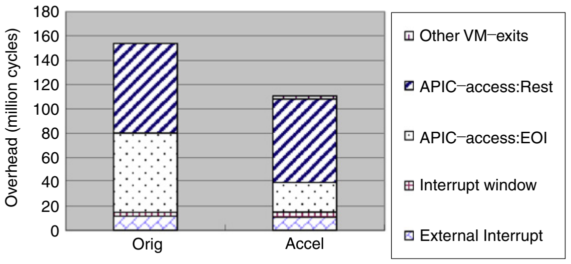

Fig. 7. Virtualization overhead per second, based on VM-exit events.

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 [31] 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 the interrupt at the 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 Fig. 6 as well, with data labeled as *n*-VM-opt. 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 the 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 in the kernel, which is in 2.6.18. It was removed later on, to avoid the runtime cost, since the operation itself is very expensive, even in the native environment.

5.2. Virtual EOI acceleration

The OS writes an end-of-interrupt (EOI) [9] 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 [9] 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 [9,31] events in Xen, to measure the CPU cycles spent, from the beginning of the VM-exit to the end, in hypervisor with the same configuration demonstrated in Section 5.1, shows that APIC-access VM-exit is the top performance bottleneck, as shown in Fig. 7. 139M cycles, or 90% of total virtualization overhead, are spent in APIC-access VM-exit for a 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, processing 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 [9] 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 an 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. 28% of total virtualization overhead is observed to be reduced, as shown in Fig. 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 did not notice any commercial OS that used this kind of *complex instruction* in practice. We argue that we do not need to worry 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.

5.3. Adaptive interrupt coalescing

High frequency interrupts can cause serious performance problems, 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 [24,7,12]. Technologies to moderate interrupts exist, such as NAPI [27], hybrid mode of interrupt disabling and enabling (DE) [26], and interrupt coalescing in modern NIC drivers, such as the IGB driver, which throttles interrupts after a certain amount of time [10,18].

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, potentially confusing the device driver. Reducing interrupt frequency can minimize virtualization overhead, but it may increase network latency, hurting TCP throughput [36]. Longer interrupt intervals 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 without overflowing the socket, application buffer, or 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* mean the number of buffers used by the application and the device driver, while t_d and *pps* mean 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, that is 64 *ap_bufs* (120832 B socket buffer size in RHEL5U1) and 1024 *dd_bufs*. An approximately 20% hypervisor intervention overhead is estimated, that is $r = 1.2$, and *pps* is sampled per second, to adaptively adjust *IF*.

Figs. 8 and 9 show the results of bandwidth and CPU utilization vs. different interrupt coalescing policies for both UDP_STREAM and TCP_STREAM: 20, 2 kHz, AIC, and 1 kHz, running 64 bit Linux 2.6.28 in an 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, 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.

In the meantime, AIC improves inter-VM communication performance, by avoiding packet loss, caused by insufficient receiving buffers. Fig. 10 shows the throughput and CPU utilization for the case where domain 0 sends packets to the 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 the 2 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 20 kHz case, or throughput drop, in the cases of 2 and 1 kHz.

6. Experiment

Incorporating the above optimizations, this section evaluates the performance and scalability of the generic virtualization architecture for SR-IOV-capable devices with aggregate 10 Gbps configuration, except Section 6.3 which uses the single port environment. 64 bit Linux 2.6.28 is used in the hardware virtual machine (HVM) container to take advantage of tickless idle, except where noted.

6.1. Experimental setup

Due to the unavailability of 10 Gbps SR-IOV-capable NIC at the time we started the research, we use ten port Gigabit SR-IOV-capable Intel 82 576 NICs in Section 6. For simplicity, the

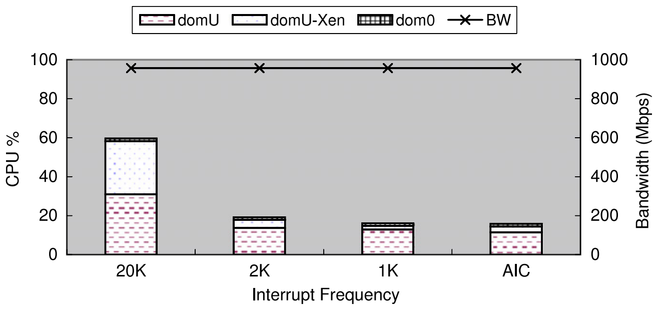

Fig. 8. Adaptive interrupt coalescing reduces CPU overhead for UDP_STREAM.

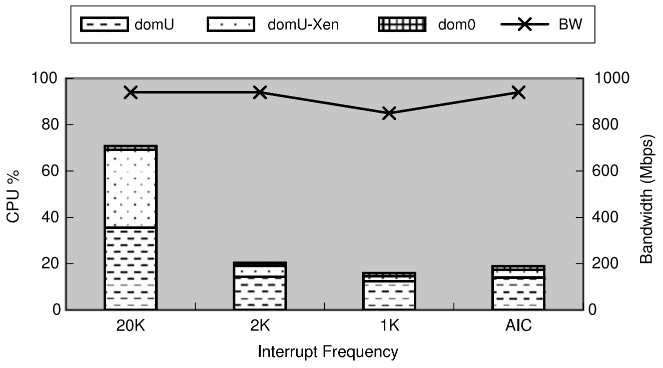

Fig. 9. Adaptive interrupt coalescing maintains throughput with minimal CPU utilization for TCP_STREAM.

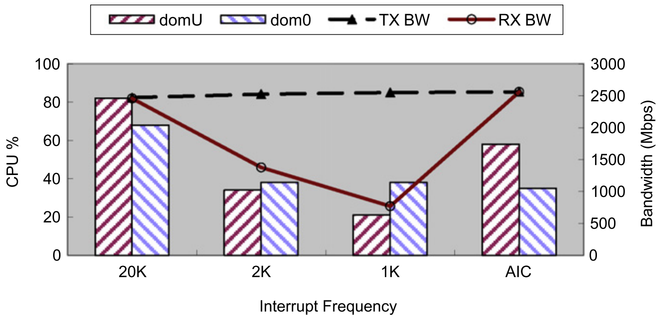

Fig. 10. Adaptive interrupt coalescing avoids packets loss in inter-VM communication.

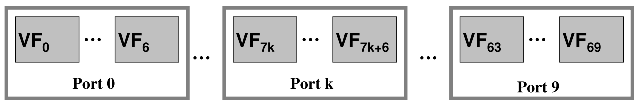

Fig. 11. VF and port allocation.

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 noted.

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 82 576 Gigabit NICs are used, to provide aggregate 10 Gbps bandwidth. Each port is configured to have 7 Virtual Functions (VFs) enabled, as shown in Fig. 11. When $10 * n$ VMs are employed in Section 6.4 of Section 6, the assigned VFs will come from VF_{7j+0} to VF_{7j+n-1} for each port j.

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 a different core, and the guest runs with only one VCPU, which is bounded evenly to the remaining threads. IGB driver 1.3.21.5 is run as a Physical Function (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 a 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.

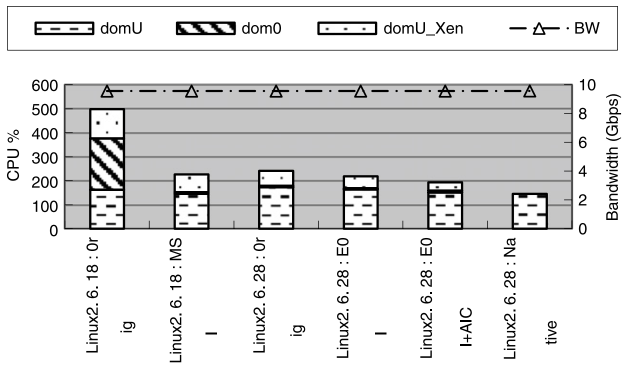

Fig. 12. Impact of the optimizations for SR-IOV with aggregate 10 Gbps Ethernet.

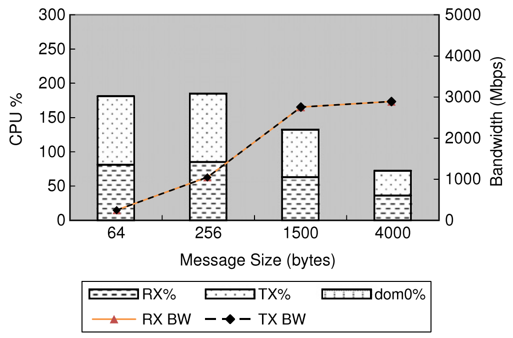

Fig. 13. SR-IOV inter-VM communication.

6.2. SR-IOV network performance

Fig. 12 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.

6.3. Inter-VM communication performance

Packets of inter-VM communication in SR-IOV are internally switched in NIC, without going through the physical line, enabling them to exceed the full line rate. Up to 2.8 Gbps throughput is achieved in SR-IOV with a single port, as shown in Fig. 13. The PV

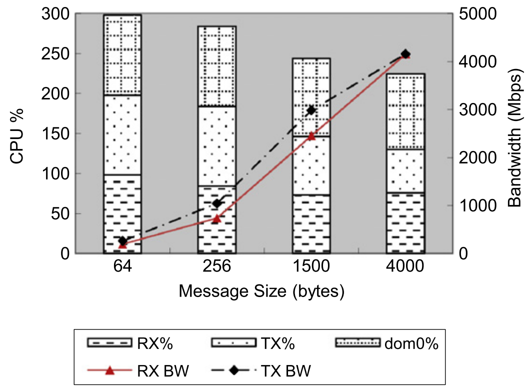

Fig. 14. PV NIC inter-VM communication.

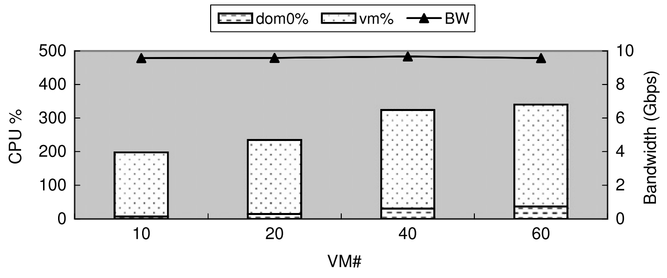

Fig. 15. SR-IOV scalability in HVM.

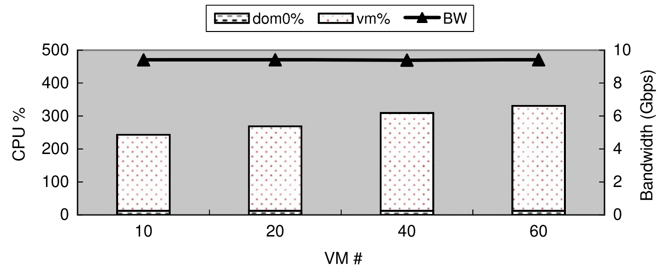

Fig. 16. SR-IOV scalability in PVM.

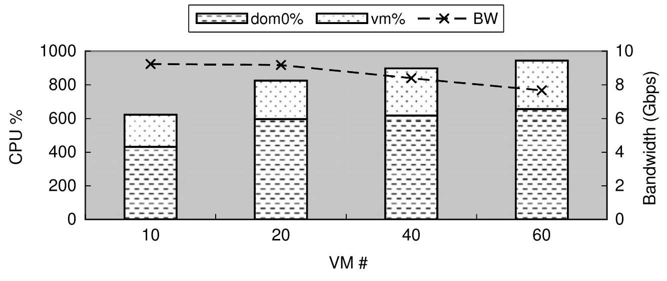

Fig. 17. PV NIC scalability in HVM.

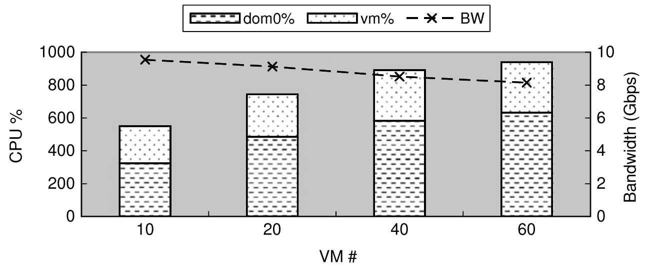

Fig. 18. PV NIC scalability in PVM.

counterpart achieves 4.3 Gbps with more CPU utilization, as shown in Fig. 14.

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, achieving higher throughput than SR-IOV. As the message size goes up from 1500 to 4000 bytes, each system call consumes more data, spending less overhead in the network stack, leading to higher bandwidth, as well. However, in terms of throughput per CPU utilization, SR-IOV is better.

6.4. Scalability

SR-IOV has almost perfect I/O scalability for both HVM and paravirtualized virtual machine (PVM), as shown in Figs. 15 and 16. 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, in the case of 10 VMs, PVM consumed slightly more CPU cycles than HVM. 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 [19].

6.5. 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 an enhanced PV NIC driver are shown in Figs. 17 and 18 for HVM and PVM, respectively. The CPU utilization goes up and the throughput drops in both cases when the VM number increases. The domain 0 CPU utilization in HVM's case is slightly higher than that in PVM's 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's case consumed slightly more CPU cycles than in HVM's case, due to the hypervisor involvement and page table switch for each system call in X86-64 XenLinux, like in Section 6.4.

6.6. Comparing with VMDq

Fig. 19 shows the scalability of VMDq [28]. The NIC we used for SR-IOV does not have VMDq support in the driver, so we used the Intel 82 598 10 Gbps NIC, for comparison. We used the Xen VMDq public tree[1] because it was the only one we could get to work stably, but it may not have the latest optimizations because

[1] http://xenbits.xensource.com/ext/netchannel2/xen-unstable.hg, http://xenbits.xensource.com/ext/netchannel2/linux-2.6.18.hg.

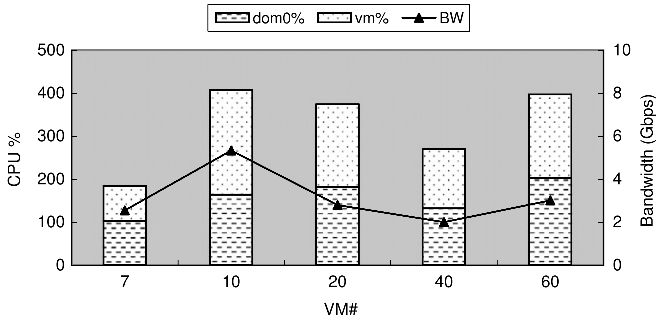

Fig. 19. VMDq scalability in PVM.

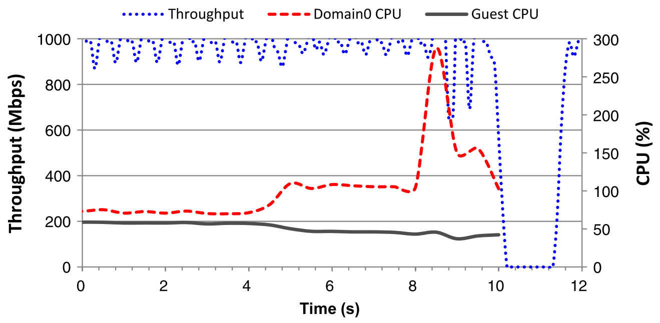

Fig. 20. Migrating an HVM running netperf with a PV network driver.

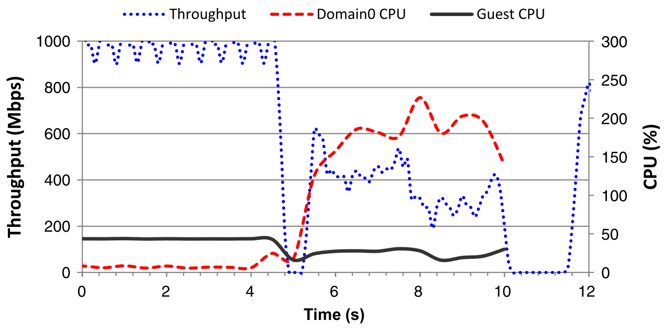

Fig. 21. Migrating an HVM running netperf with SR-IOV and DNIS.

the tree has been inactive since May 9 and the results are not very stable. The results show the VMDq performance reaches its peak at 10 VMs, and drops progressively as the VM# increases. This is probably because the NIC has only 8 queue pairs, and only 7 guests can get VMDq support. Once the VM# exceeds 7, the rest of the VMs share the network with domain 0, as the conventional PV NIC driver does, which sacrifices CPU cycles to copy the packets, but achieves additional throughput. The interesting thing is that the throughput goes up when the VM# increases from 40 to 60. We suspect it may be caused by a program defect in the tree.

6.7. Migration

Figs. 20 and 21 show the migration characteristics of the PV network driver and SR-IOV, with dynamic network interface (DNIS) in 1 Gbps network configuration, that is with single port 82 576 NIC. Before migration, SR-IOV completely eliminates CPU utilization in domain 0, while PV network uses significant CPU cycles to service the network requests from the guest. The migration starts at 4.5th second for both cases. The DNIS incurs an additional step of network interface switch and virtual hot removal of the VF device on the guest side, incurring an additional 0.6 s service shutdown time at very beginning of migration, due to packet loss at interface switch time.

The PV network driver suffers the service shuts down at 10.4th second, due to the VM state copy and transportation in the last step. The service in PV network is restored at 11.8th second at target platform. DNIS undergoes a similar situation for the “real” migration. The service shuts down at 10.3th second and restores at 11.8th second, on par with the PV network driver.

7. Conclusion

We designed, implemented, and tuned a generic virtualization architecture for an SR-IOV-capable network device. The architecture supports reusability of PF and VF drivers across different VMMs, and also implements dynamic network interface switching to facilitate migration. 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 measurements to compare the SR-IOV solution with others. Our experimental data proved that SR-IOV provides a good solution for high performance I/O virtualization, without sacrificing migration.

As CPU computing capability continues to scale with Moore’s Law, it is 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.

Acknowledgments

The other members contributing to this study include 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. This work was also supported by the National Natural Science Foundation of China (Grant Nos. 60970108, and 60970107), Ministry of Science and Technology of China (Grant No. 2011DFA10850), the Science and Technology Commission of Shanghai Municipality (Grant Nos. 10DZ1500200, and 10511500102).

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, 2003, pp. 164–177.
[2] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, Andrew Warfield, Live migration of virtual machines, in: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, May 02–04, 2005, pp. 273–286.
[3] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Remus: high availability via asynchronous virtual machine replication, in: Proceedings of the 5th Conference on Symposium on Networked Systems Design & Implementation, NSDI 08, 2008.
[4] T. Davis, W. Tarreau, C. Gavrilov, C.N. Tindel, Linux Ethernet Bonding Driver, Linux How to Documentation, April, 2006.
[5] Y. Dong, et al. High Performance Network Virtualization with SR-IOV, HPCA-16, Bangalore, India, 2010.
[6] Y. Dong, J. Dai, et al. Towards high-quality I/O virtualization, in: Proceeding of the Israeli Experimental Systems Conference, SYSTOR, Haifa, Israel, 2009.
[7] Y. Dong, D. Xu, Y. Zhang, G. Liao, Optimizing network virtualization with virtual interrupt coalescing and receive side scaling, IEEE Cluster, 2011.
[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, 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.

[9] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developers' Manual. http://www.intel.com/products/processor/manuals/index.htm.
[10] Intel Corporation, Intel® 82576 Gigabit Ethernet Controller Datasheet. http://www.intel.com.
[11] Kernel Virtual Machine. http://kvm.qumranet.com/.
[12] G. Liao, L. Bhuyan, Performance measurement of an integrated NIC architecture with 10GbE, in: 17th Symposium on High-Performance Interconnects (HOTI), USA, 2009.
[13] G. Liao, D. Guo, L. Bhuyan, S.R. King, Software techniques to improve virtualized I/O performance on multi-core systems, in: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, San Jose, CA, 2008, pp. 161–170.
[14] G. Liao, X. Zhu, L. Bhuyan, A new server I/O architecture for high speed networks, in: Proceeding of the 17th IEEE International Symposium on High Performance Computer Architecture, San Antonio, Texas, 2011.
[15] G. Liao, X. Zhu, S. Larsen, et al. Understanding power efficiency of TCP/IP packet processing over 10GbE, in: 18th Symposium on High-Performance Interconnects (HOTI), Mountain View, CA, 2010.
[16] J. Liu, et al. High Performance VMM-Bypass I/O in Virtual Machines, in: Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2006, p. 29–42.
[17] A. Menon, A.L. Cox, W. Zwaenepoel, Optimizing Network Virtualization in Xen, in: Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2006, pp. 15–28.
[18] J.C. Mogul, K.K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel, ACM Transactions on Computer Systems TOCS 15 (3) (1997) 217–252.
[19] J. Nakajima, A. Mallick, X86-64 XenLinux: architecture, implementation, and optimizations, in: Proceeding of Ottawa Linux Symposium, OLS, 2006.
[20] D. Ongaro, A.L. Cox, S. Rixner, Scheduling I/O in virtual machine monitors, in: Proceedings of 4th ACM/USENIX International Conference on Virtual Execution Environments, Seattle, WA, 2008, pp. 1–10.
[21] PCI Special Interest Group. http://www.pcisig.com/home.
[22] H. Raj, K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices, in: Proceedings of the 16th International Symposium on High Performance Distributed Computing, Monterrey, CA, 2007.
[23] K.K. Ram, J.R. Santos, Y. Turner, A.L. Cox, S. Rixner, Achieving 10 Gb/s using safe and transparent network interface virtualization, in: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Washington, DC, 2009.
[24] K. Ramakrishnan, Performance consideration in designing network interfaces, IEEE Journal on Selected Areas in Communications 11 (2) (1993) 203–219.
[25] M. Rosenblum, T. Garfinkel, Virtual Machine Monitors: Current technology and future trends, Computer 38 (5) (2005) 39–47.
[26] K. Salah, A. Qahtan, Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme, Computer Communications 32 (1) (2009) 179–188.
[27] J. Salim, When NAPI comes to town, in: Proceedings of Linux 2005 Conference, Swansea, UK, August 2005.
[28] J.R. Santos, Y. Turner, G. Janakiraman, I. Pratt, Bridging the gap between software and hardware techniques for I/O virtualization, in: Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2008, pp. 29–42.
[29] M. Steil, Inside VMware, 2006. http://events.ccc.de/congress/2006/Fahrplan/attachments/1132-InsideVMware.pdf.
[30] J. Sugerman, G. Venkitachalam, B. Lim, Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor, in: Proceedings of the General Track: 2002 USENIX Annual Technical Conference, Boston, MA, 2001, pp. 1–14.
[31] 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) (2005) 48–56.
[32] W. Voorsluys, et al. Cost of virtual machine live migration in clouds: a Performance evaluation, in: Proceedings of the 1st International Conference on Cloud Computing, Beijing, China, 2009.
[33] J. Wang, K. Wright, 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, pp. 109–118.
[34] P. Willmann, S. Rixner, A.L. Cox, Protection strategies for direct access to virtualized I/O devices, in: Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2008, 15-28.
[35] A. Whitaker, M. Shaw, S.D. Gribble, Denali: lightweight virtual machines for distributed and networked applications, Technical Report 02-02-01, University of Washington, 2002.
[36] 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) (2005) 434–445.
[37] E. Zhai, G.D. Cummings, Y. Dong, Live migration with pass-through device for linux vm, in: Ottawa Linux Symposium, OLS, 2008.

Yaozu Dong is a senior staff in Intel Open Source Technology Center. His research focuses on architecture and system including virtualization, operating system, and distributed and parallel computing.

Xiaowei Yang was a senior software engineer in Intel Open Source Technology Center. His research interests include virtualization performance and scalability, operating system and clusters. Xiaowei Yang received his master's degree in computer science and engineering from Shanghai Jiao tong University, in 2006.

Jianhui Li is a software architect in software and service group of Intel. His interests include binary translation, performance, and virtualization. He received a Ph.D. degree in computer science from Fudan University (China).

Guangdeng Liao received the B.E. and M.E. degrees in computer science and engineering from Shanghai Jiao tong University, and the Ph.D. degree in computer science from the University of California, Riverside. He currently works as a research scientist, at Intel Labs. His research interests include I/O architecture, operating system, virtualization and data centers.

Kun Tian received his M.E. degree in communication system from University of Electronic Science and Technology of China. He is now in Intel Open Source Technology Center, with the research focus on system architecture.

Haibing Guan received his Ph.D. degree in computer science from the TongJi University (China), in 1999. He is currently a professor of Computer Science, Shanghai Jiao Tong University (Shanghai, China). His current research interests include, but are not limited to, computer architecture, compiling, virtualization and hardware/software co-design.