Network Interface Virtualization: Challenges and Solutions

Ryan Shea and Jiangchuan Liu, Simon Fraser University

Abstract

Recently, computer virtualization has become an important technology in nearly every large organization's IT infrastructure. At its core, computer virtualization allows a single physical machine to be split into several virtual machines. Through the use of VMs, organizations can decrease operational costs through server consolidation and better utilization of computational resources. VMs also allow organizations to outsource their computational requirements to utility computing platforms such as cloud computing. The benefit of computer virtualization comes together with overhead, affecting virtually every subsystem, particularly the network subsystem that involves both hardware and software components, and both local and remote accesses. This article provides an intuitive understanding of state-of-the-art software-based virtual network interfaces and the causes of virtualization overhead through both system analysis and real experiments. We further discuss the recent hardware advances in this field as well as the challenges yet to be addressed.

From remote login and server consolidation to cloud computing, *computer virtualization* has fast become a indispensable technology in today's modern information technology infrastructure. In its simplest form, computer virtualization can be described as the ability to host many virtual machines (VMs) on a single physical machine. Recent surveys have shown that 90 percent of large organizations use computer virtualization in some capacity in their IT infrastructure [1]. The rapid uptake of this technology has been fueled by promises of increased resource utilization, server consolidation, and increased flexibility [2]. It has also made the long-held dream of public utility computing platforms a reality. One prominent utility computing example is Amazon's EC2, which went public in 2006 and is now estimated to generate $500 million in revenue per year. It is now clear computer virtualization has been a success and is here to stay. However, as with any new technology, there are pros and cons, and computer virtualization is no exception. Running multiple VMs on a single physical machine inevitably creates significant overhead; and, not surprisingly, almost every computer subsystem is affected to some degree. The networking interfaces, however, can suffer some of the worst overhead given its complicated operations and interactions with other subsystems. Often, a virtualized networking interface can consume several times the number of processor cycles and memory accesses to deliver a single packet than a non-virtualized interface.

The problem of high overhead in virtualized networking interfaces has been a topic of interest for both academia and industry. This interest leads to new software methods as well as hardware devices in an attempt to lower the overhead. Although much success has been achieved in alleviating the overhead, certain traffic flows in virtual machines continue to consume far more resources than their physical counterparts, making network interfaces among the most difficult subsystems to virtualize.

This article provides an intuitive understanding of state-of-the-art software-based virtual network interfaces. We also seek to identify the root causes of virtualization overhead through both system analysis and real experiments. We further discuss the recent hardware advances that facilitate virtual network interface design as well as the challenges yet to be addressed. Next, we briefly introduce computer virtualization and give a short history.

Virtualization: An Overview

Any virtualization system must ensure that each VM be given fair and secure access to the underlying hardware. In state-of-the-art virtualization systems, this is often achieved through the use of a software module known as a *hypervisor*, which works as an arbiter between a VM's virtual devices and the underlying physical devices. The use of a hypervisor brings many advantages, such as device sharing, performance isolation, and security between running VMs. Having to consult the hypervisor each time a VM makes a privileged call, however, introduces considerable overhead as the hypervisor must be brought online to process each request. Figure 1 shows the interactions between three different VMs and the hypervisor. The solutions to full computer virtualization can be broadly split into two categories, as described below.

Paravirtualization Machines

Paravirtualization machines (PVMs) were the first form of full computer virtualization and are still widely deployed today. The roots of paravirtualization run very deep indeed with the first production system known as VM/370[1] [3], created by IBM and available in 1972, many years before paravirtualiza-

[1] *Since IBM also created a CPU instruction set specifically for VM/370, it is arguably not a pure PVM system.*

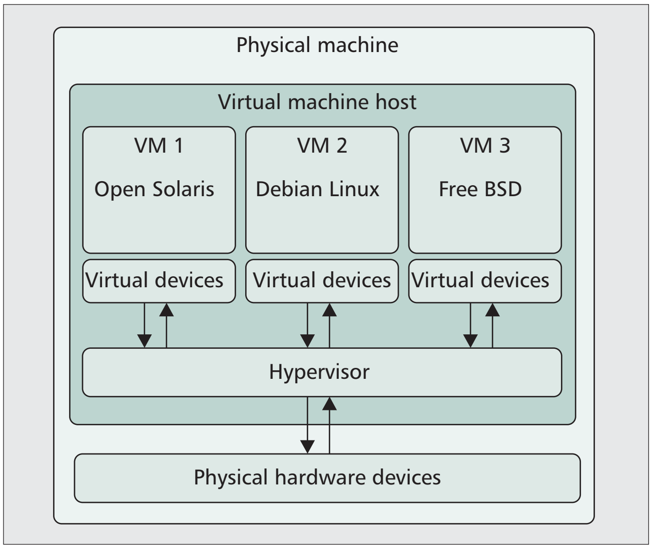

Figure 1. *Hypervisor-based computer virtualization.*

tion became a mainstream product. The VM/370 was a multiuser system which provided the illusion that each user had their own operating system. Paravirtualization requires no special hardware and is implemented by modifications to the VM's operating system. The modifications instruct the operating system (OS) to access hardware and make privileged system calls through the hypervisor; any attempt to circumvent the hypervisor will result in the request being denied. Modifying the OS does not usually create a barrier for open source OSs such as Linux; however, proprietary OSs such as Microsoft Windows can pose a considerable challenge. The flagship example of a PVM system is Xen,[2] which first became an open source project in 2002. The Xen Hypervisor is also a keystone technology in Amazon's successful cloud service EC2.

Hardware Virtual Machines

Hardware virtual machines (HVMs) rely on special hardware to achieve computer virtualization. An HVM works by intercepting privileged calls from a VM and handing these calls to the hypervisor. The hypervisor decides how to handle the call, ensuring security, fairness, and isolation between running VMs. The use of hardware to trap privileged calls from the VMs allows multiple unmodified OSs to run. This provides tremendous flexibility as system administrators can now run both proprietary and legacy OSs in the VM. In 2005, the first HVM compatible CPU became available, and as of 2012 nearly all server-class and most desktop-class CPUs support HVM extensions. Both Intel and AMD implement HVM extensions, referred to as Intel VT-X and AMD-V, respectively. Since HVMs must intercept each privileged call, considerably higher overhead can be experienced than with PVMs. This overhead can be especially high when dealing with input/output (I/O) devices such as the network card, a problem that has led to the creation of paravirtualization drivers. Paravirtualization drivers such as the VirtIO package for KVM allow a VM to reap the benefits of HVMs such as an unmodified OS while mitigating much of the overhead [4]. Examples of HVM-based virtualization systems include KVM[3] and VMware[4] servers.

Figure 2 shows the simplified architecture of both hardware virtualization and paravirtualization. It is important to note that PVM and HVM are not mutually exclusive and often overlap, borrowing aspects from each other. For example, the PVM hypervisor Xen can use hardware virtualization extensions to host proprietary OSs such as Microsoft Windows. The use of the VirtIO paravirtualization drivers by KVM is an example of HVM using a hybrid approach to virtualization.

There is also a third class of virtualization know as container virtualization or OS virtualization, which allows each user to have a secure container and run their own programs in it without interference. It has been shown to have the lowest overhead when compared to PVMs and HVMs [5, 6]. This low overhead is achieved through the use of a single kernel shared between all containers. Such a shared kernel does have significant drawbacks, however, in that all users must use the same OS. For many architectures such as public utility computing, container virtualization may not be applicable as each individual user wants to use their own operating system in their VM, and is therefore not the focus of this article.

Virtualizing Devices

Many important operations in virtualized systems suffer from some degree of virtualization overhead. For example, in both PVM and HVM, each time a VM encounters a memory page fault the hypervisor must be brought on the CPU to rectify the situation. Each of these page faults consists of several context switches, as the user space process is switched for the guest kernel, the kernel for the hypervisor, and sometimes the hypervisor for the host kernel. Compare this to a bare-metal operating system that generally has only two context switches, the user space process for the kernel process and back again. Disk access has similar problems. It is fairly intuitive that the higher number of context switches and their associated operations can impart considerable overhead on privileged calls as each call now consumes considerably more CPU cycles to complete. However, as stated earlier, the hypervisor is a necessary evil as it is required to operate as an arbiter between running VMs and the hardware for both performance isolation and security reasons.

Why Is the NIC More Problematic to Virtualize?

To understand why the network interface controller (NIC) is even more difficult to virtualize, we now briefly compare and contrast the network subsystem to other subsystems. The memory subsystem, for example, has arguably already been well virtualized. Through the use of virtual memory, modern operating systems allow a process to allocate and use memory without regards to other process; the OS along with a piece of hardware known as a translation look-aside buffer (TLB) converts the virtual address to a physical one. A process could use its assigned memory with assurances from the OS that the memory would not be accessed or changed by other processes without permission. Full computer virtualization requires an additional translation from the VM's "physical" address space to the host's actual physical address space. In both PVM and HVM, the hypervisor plays an integral role in address translation; however, HVM also takes advantage of recent hardware advances that allow the translations to be stored in hardware, reducing the number of calls to the hypervisor. In a sense the existing virtual memory systems have been fully virtualized; this was not an easy feat, but there was considerable architectural knowledge on how to make it a reality.

Network interface virtualization, on the other hand, still lacks a rich and well developed set of techniques that allow many processes to securely and directly access the device with

[2] *http://www.xen.org*

[3] *http://www.linux-kvm.org/*

[4] *http://www.vmware.com/*

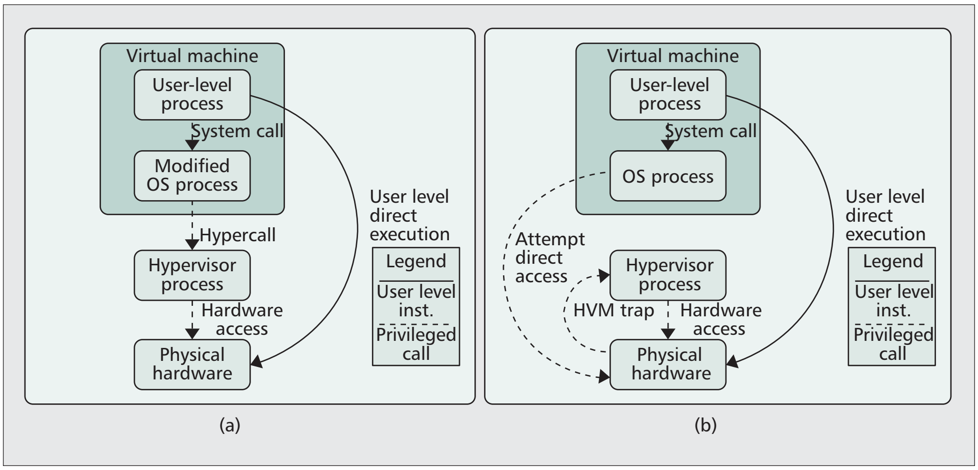

Figure 2. *PVM and HVM simplified architecture: a) PVM; b) HVM.*

performance or security guarantees. Through the use of IP addresses and port numbers, we have to rely on the many layers of the network stack to keep processes from interfering with each other. With virtualization, we have duplication in the network stack, and a packet must travel through this duplication, which can cause a single packet to traverse many different processes before reaching its destination.

Another major difference between networks and other subsystems is in how data is retrieved from the devices. Both the file system and memory subsystem are pull interfaces, through which data generally only arrives when a request is made from within the VM. Since the requests originate from within the VM, we have the ability to control the priority of the request as well as some knowledge on how long the data access will take. The network subsystem, however, is in a starkly different situation. Sending is quite similar since this activity is initiated from inside the VM. Receiving, on the other hand, is initiated by other hosts in the network, and a packet could show up at anytime and be of any priority (including unsolicited or malicious). On receipt of a packet, the physical network interface sends an interrupt, which causes the hypervisor to be scheduled to run on a processing core. If the physical machine is fully utilized, a running VM must be interrupted so that the hypervisor can deliver the packet. In the next section, we discuss in detail how the latest hypervisor designs handle these issues and deliver packets to the VMs.

Virtual Network Interfaces: Implementation and Performance

We now focus on the design of efficient virtual NICs, which can be roughly classified into two categories: software-based and hardware-assisted interfaces. Software-based interfaces have been around since computer virtualization first became a reality; however, they tend to suffer from higher overhead. Recently, hardware manufacturers have become aware of the need to lower virtual interface overhead and, as of 2009, we have seen that devices can alleviate some of the overhead.

Software-Based Virtual Interfaces

Much research has been done on virtual network interfaces to make them as efficient as possible. Software interfaces are generally considered to be in one of two classes: emulated or paravirtualized. Emulated interfaces are useful for their great compatibility and flexibility. Popular choices for emulation include Realtek rtl8139 and Intel E1000. Any OS with drivers for the emulated interface can simply use it as if it were a physical interface. This provides great flexibility to run legacy OSs and OSs that require specific hardware. To emulate the interface a hypervisor must use hardware virtualization extensions to trap each attempt to access the device. For example, when a VM wants to send on the emulated interface, the hypervisor is scheduled to run, so it can process the request and emulate the hardware response. Each attempt to access the hardware creates a VM exit, which in a highly utilized network can be disastrous to the VM's performance. For this reason, the emulated interfaces are only used if absolutely necessary and are rarely seen in high-performance applications.

The second class of software virtual interfaces is from paravirtualization, which are devices the guest OS knows are virtualized. Paravirtualization offers superior performance to emulated interfaces, because the VM can have more control over when a VM exit is created due to an interface request. For example, we can bundle several packets to be sent into a single VM exit and reduce the number of context switches between the hypervisor and the VM. Receiving and sending multiple packets at once also allows for more efficient use of the processor's last level cache (LLC), as more operations are done sequentially. There are also significant savings in terms of processor cycles since paravirtualization drivers need not emulate unnecessary hardware features. All major virtualization systems implement paravirtualization drivers to increase performance including Xen, VMware server, and KVM.

Micro Experiment One: Bare-Metal, KVM VirtIO, and KVM Emulated rtl8139 Network Overhead

The best way to show the difference in overhead between bare-metal interfaces and the different implementations of software-based virtual interfaces is a small experiment. To this end, we designed an experiment to measure the overhead of sending and receiving on different interfaces.

We compiled KVM 0.15.1 from source and installed it in our test system. Our test system was a modern mid-range PC with an Intel Core 2 Q9500 quad core processor running at 2.83 GHz. We enabled Intel VT-X in the BIOS as it is required for HVM support. The PC was equipped with 4 Gbytes of 1333 MHz DDR-3 SDRAM and a 320 Gbyte 7200 RPM hard drive with 16 Mbytes cache. The physical network interface is a 1000 Mb/s Broadcom Ethernet adapter attached to the PCI-E bus. The host and KVM guest used Debian Squeeze as their operating system. We used the Iperf network benchmark to create the TCP traffic to and from a remote host in the same subnet. We limit the remote host to 10 Mb/s using the Ethernet configuration tool Ethtool. We use the Linux hardware performance analysis tool Perf to collect system-level statistics such as processor cycles consumed, LLC references, interrupt requests, and processor context switches. For each experiment, we instruct Perf to collect five samples each with a duration of ten seconds and then average them. We test three systems: our bare metal Linux host, KVM with paravirtualized VirtIO drivers, and KVM with an emulated rtl8139 network card. It is important to note that although we use KVM for our experiments, we expect similar results would be found with any virtualization package that relies on software-based virtual network interfaces.

	Bare-metal	KVM VirtIO	KVM rtl8139
Cycles	11.5M/s	51.8M/s	95.6M/s
LLC references	0.48M/s	2.3M/s	3.6M/s
Context switches	189/s	452/s	452/s
IRQs	600/s	2600/s	5000/s

Table 1. *Iperf receiving 10 Mb/s TCP traffic.*

	Bare-metal	KVM VirtIO	KVM rtl8139
Cycles	4.1M/s	33.4M/s	74.2M/s
LLC references	0.13M/s	1.3M/s	2.7M/s
Context switches	19/s	307/s	432/s
IRQs	180/s	1700/s	3500/s

Table 2. *Iperf sending 10 Mb/s TCP traffic.*

The results for the receiving experiment are given in Table 1, and the results for the sending experiment are given in Table 2. The CPU cycles and LLC references are given in millions per second. The context switches and interrupt requests are given in number per second. In the receiving experiment, KVM with VirtIO takes nearly five times more cycles to deliver the packets to the VM as the bare-metal host. This, however, is a huge improvement over the emulated rtl8139 interface, which takes nearly nine times more cycles to deliver the same amount of traffic to the VM. KVM with VirtIO and the emulated rtl8139 are also much less efficient on cache than the bare-metal system. This is due to the fact that the VM's hypervisor must copy each packet from the memory space of the host to the VM's space. These copies use up valuable processor cycles and can also evict data from the processor cache. VirtIO manages to be much more efficient than the emulated device because it generally copies more packets sequentially. The number of context switches increases for both the paravirtualized and emulated interfaces, because the hypervisor must be brought on a CPU in order to copy the packet into the VM's memory space. Finally, we look at interrupt requests (IRQs) that are used by the physical device to notify the kernel of an incoming packet, and by the hypervisor to indicate to a running VM that it has received packets. Since KVM also uses interrupts to indicate to a running VM that it has received packets from the network, it comes as no surprise that they would be considerably higher. In comparison to the emulated rtl8139 interface, VirtIO manages to reduce the IRQs by generally delivering more packets with a single interrupt.

We now focus on the sending system level statistics found in Table 2. As can be seen, all metrics have decreased, which is unsurprising considering sending is a more deterministic task from the point of view of the sender, as it has more control over the data flow.

The reason that software virtual interfaces have several times the overhead of bare-metal interfaces can be explained by comparing the packet and interrupt path in the system. Figure 3 shows the different paths taken by a packet in the bare-metal system and KVM. In the bare-metal system, requests for access to the network stack and device drivers are often handled by the same kernel process. Because of this, there are far fewer context switches, fewer cycles consumed, and less cache used, since the same process is responsible for processing the packet and delivering it to the application layer. KVM, however, is in a strikingly different situation, in which a packet must traverse multiple processes in order to be serviced. This is because when the packet is received on the physical device, it is copied by the host machine's kernel into the memory space of the VM; the VM is then notified of the incoming packet, and it is scheduled to run on the processor; the VM delivers the packet to its network stack, where it is finally delivered to the application layer. All these extra steps, when applied to each packet, contribute to the increase in CPU cycles, cache usage, context switches, and interrupt requests we witnessed in the micro experiment.

Hardware-Assisted Virtual Interfaces: Design and Performance

It is clear from our first micro-experiment that using software-based virtual interfaces is much more expensive than bare metal interfaces. Hardware manufactures have taken notice of this high overhead and have created devices that greatly help alleviate virtual NIC overhead. Two popular solutions are Virtual Machine Device Queues (VMDQ)[5] and Single Root I/O Virtualization (SR-IOV).[6] Both of them are referred to as VT-C by Intel, and both work by duplicating components of the NIC to give a VM more direct access to the hardware. Note, however, that SR-IOV has become an industry standard, and therefore different companies are free to package it in their network interface cards.

Virtual Machine Device Queues and Single-Root I/O Virtualization

Traditionally, a physical NIC has a single interrupt, which in a multicore system is mapped to a single core. On receipt of a packet the NIC initiates an IRQ, which schedules the kernel to run on the interrupted core. This single interrupt can cause significant overhead for virtualized systems as we describe in the following example. Imagine a situation where a two-core machine is hosting two VMs, each pinned to a specific core with VM-1 on core-1 and VM-2 on core-2, respectively. The physical NIC has its interrupt mapped to core-1. Now a packet arrives for VM-2 from the network, and the NIC sends an interrupt. VM-1 is suspended while the kernel investigates the packet; noticing the packet is for VM-2, the kernel sends an interrupt to VM-2 on core-2. Clearly, every time a packet arrives for VM-2, VM-1 is unnecessarily interrupted. The impact becomes aggravated with each VM and core being added to the system.

Virtual machine device queues (VMDQ) solves the above problem by only interrupting the VM that the packet was destined for. To do this, VMDQ contains multiple interrupts that can be assigned to different VMs. On receipt of a packet, a VMDQ enabled NIC uses a layer 2 classifier to determine to which VM the packet is destined based on either its medium access control (MAC) address or VLAN tag. The NIC then uses direct memory access (DMA) to copy the packet into the

[5] *Intel's VMDQ white paper can be found at: software.intel.com/file/1919*

[6] *More info on SR-IOV can be found at: www.intel.com/go/vtc*

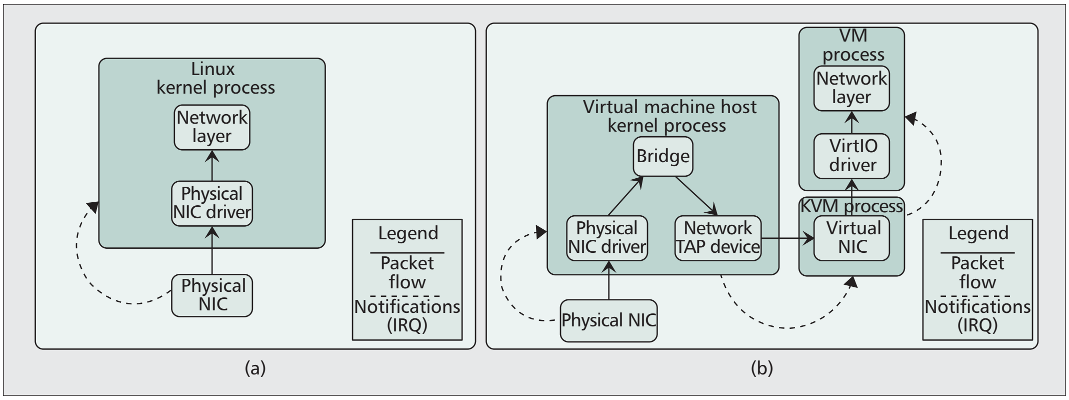

Figure 3. *Packet delivery: a) bare-metal; b) KVM.*

	Bare-metal	KVM SR-IOV	KVM VirtIO
Cycles	11.5M/s	31.1M/s	51.8M/s
LLC references	0.48M/s	2.1M/s	2.3M/s
Context switches	189/s	476/s	452/s
IRQs	600/s	1200/s	2600/s

Table 3. *Iperf receiving 10 Mb/s TCP traffic.*

	Bare-metal	KVM SR-IOV	KVM VirtIO
Cycles	4.1M/s	18.8M/s	33.4M/s
LLC references	0.13M/s	0.72M/s	1.3M/s
Context switches	19/s	239/s	307/s
IRQs	180/s	390/s	1700/s

Table 4. *Iperf sending 10 Mb/s TCP traffic.*

host's memory, and sends an interrupt to the destination VM. The hypervisor then collects the packet, copies it into the VM memory space and notifies the VM.

Although VMDQ goes a long way to solve some of the overhead and scalability associated with virtual NICs the hypervisor must still copy the packet from the host's memory space into the VM space. To solve this, hardware manufacturers have standardized single-root I/O virtualization (SR-IOV), which, when applied to network cards, removes the extra packet copy. To accomplish this, an SR-IOV enabled NIC not only can be configured to send an interrupt to the proper core, but also DMAs the packet directly into the VM's memory space. Using SR-IOV, VMs can not only save considerable overhead but also hit prior unattainable speeds such as 10 Gb/s [7]. Figure 4 shows how packets are delivered from the network to VMs using both VMDQ and SR-IOV.

Micro Experiment Two: Bare-Metal, KVM SR-IOV, and KVM VirtIO Network Overhead

Hardware-assisted virtual NICs add many new features that help alleviate substantial overhead; however, they do not solve the issue outright. For example, due to a limitation in the x86 architecture, the hypervisor must still be involved in handling IRQs generated by a VMDQ or SR-IOV enabled NIC. These interrupts must be handled by the hypervisor due to security issues related to directly mapping the interrupt into a guest VM. This security limitation as well as overhead created by mapping the devices into the VM result in the overhead still being considerably higher even with these devices deployed.

To show the performance overhead of these hardware-assisted virtual NICs we devised another micro experiment to compare SR-IOV, bare-metal, and paravirtualized devices in terms of overhead. Once again we use KVM and compare its performance while using the paravirtualization drivers VirtIO and an SR-IOV enabled guest against the optimal bare-metal performance. We chose to test SR-IOV over VMDQ for two important reasons: first, SR-IOV has lower overhead than VMDQ since SR-IOV includes all the features of VMDQ plus additional support for sending the packets directly into a virtual machine's memory space using DMA; second, KVM does not appear to have stable support for VMDQ. To provide SR-IOV capabilities to KVM we used an Intel I350 gigabit server adapter. Once again we measured the performance of our virtual machine while it receives and sends a 10 Mb/s TCP data stream. Once again we measure system level performance statistics such as CPU references, context switches, and IRQs for our SR-IOV enabled virtual machine using the same methods described in micro experiment one.

Table 3 gives the results for receiving data and Table 4 the results for sending. As can be seen from both tables, SR-IOV greatly reduces the overhead of the network traffic in nearly every metric when compared to VirtIO. For both sending and receiving, hardware assisted SR-IOV consumes approximately 40 percent less cycles than the paravirtualized driver VirtIO. However, when compared to the system's bare-metal performance SR-IOV still consumes several times more processor cycles. When compared to VirtIO the cache performance of SR-IOV is also improved since the packets are now directly copied into the VM's memory space. In terms of processor context switches SR-IOV shows similar results to VirtIO. As described previously, the hypervisor must still be switched on the processor to handle IRQs generated by the SR-IOV enabled NIC, and this is likely why we do not see a great improvement in terms of the numbers of context switches. Finally, in terms of both sending and receiving SR-IOV greatly reduces the number of IRQs generated by the VM. This is expected as now the interrupt can be directly handled by the VM's hypervisor.

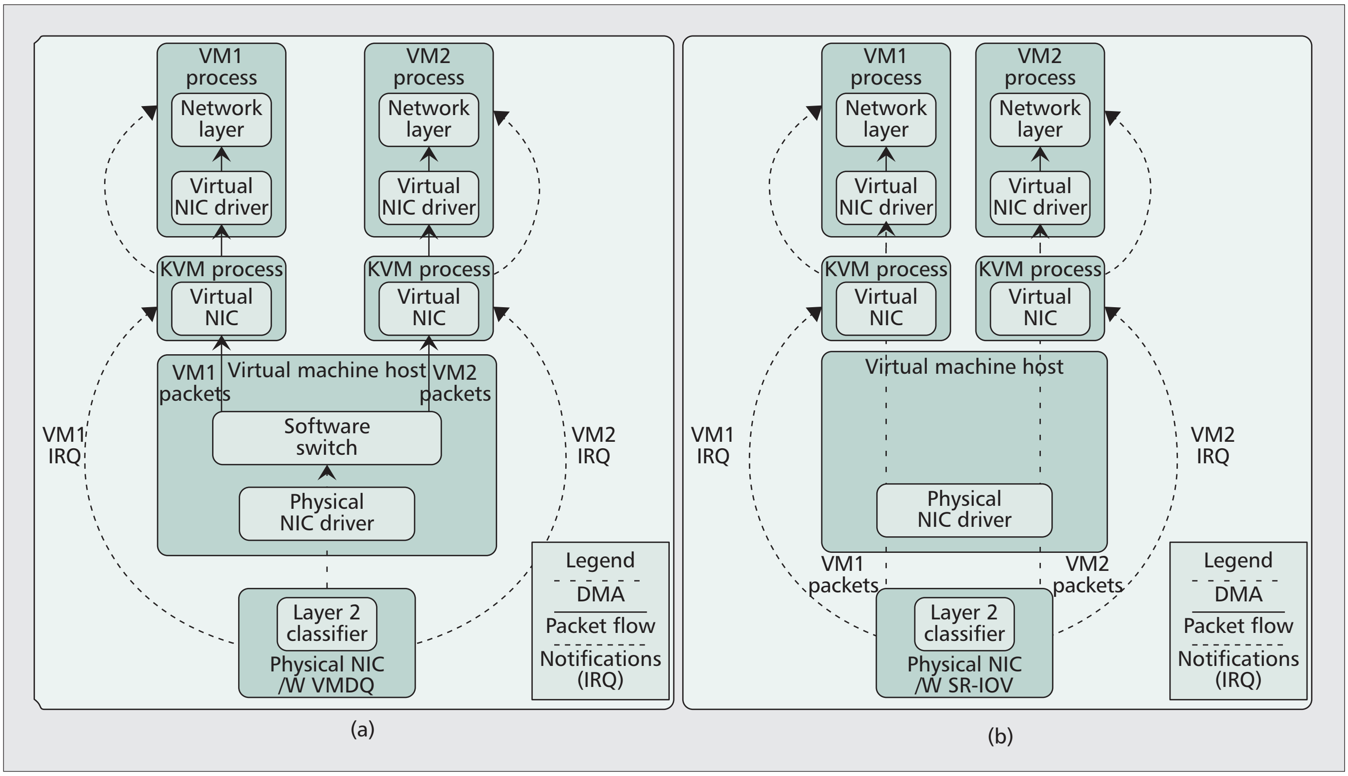

Figure 4. *a) VMDQ; b) SR-IOV.*

In summary, we can see that the use of hardware devices can greatly decrease the amount of overhead experienced by a virtual network interface; however, there is still much room for improvement since in all our tested metrics SR-IOV still consumes several times the amount of resources compared to a bare-metal network interface.

Concluding Remarks and Future Directions

Although computer virtualization is deployed in some capacity in nearly every large institution, the inherent virtualization overhead is still a considerable obstacle for many applications. From large-scale scientific computing using the Message Passing Interface (MPI) to a busy web server, organizations must weigh the benefits of virtualization against the performance penalty due to overhead. The number of applications that can benefit from computer virtualization will clearly increase with more efficient solutions being developed.

We have shown that the network interface is one of the most difficult computer subsystems to virtualize. Despite great strides in both academia and industry, there remain many challenges to face. The heavy use of virtualized interfaces in cloud computing has also introduced some issues in terms of network delay and bandwidth stability [8]. The heavy virtualization overhead experienced by the network and memory subsystem can create a barrier for scientific computing applications that wish to use a cloud platform for their computing [9]. Even with advanced hardware technologies like SR-IOV, we have shown that virtual NICs still generate measurably more overhead than their physical counterparts. A considerable amount of the overhead in SR-IOV is due to the fact that the hypervisor must still handle IRQs generated by the SR-IOV enabled NIC. Recently, techniques have been developed to alleviate some of the overhead created by these IRQs by allowing a VM to have direct control over the interrupt system [10]. There are also scalability issues as SR-IOV and VMDQ have upper limits on how many VMs can use the advanced features at a given time.

Security is another concern with virtualized computers and network interfaces, particularly in the case of networked denial of service (DoS) attacks on virtualized systems. We have performed some initial experiments on the effects of DoS attacks on virtual machines, and it appears that due to virtualization overhead, a DoS attack on a VM can consume almost 10 times more resources than on a bare metal machine [11]. Our findings also indicate that such previously successful endpoint TCP DoS mitigation strategies as SYN cookies and SYN caches are much less effective at protecting VMs than bare-metal systems. It is therefore necessary to develop robust solutions, for example, through hardening the paravirtualization drivers against DoS attacks and deploying defenses such as SYN-proxies in the hypervisor.

Acknowledgment

This work is supported by an NSERC Discovery Grant and an NSFC Major Program of International Cooperation Fund.

References

[1] CDW. Server Virtualization Life Cycle Report, Jan. 2010.
[2] W. Vogels, "Beyond Server Consolidation," *Queue*, 6:20–26, Jan. 2008.
[3] R. J. Creasy, "The Origin of the vm/370 Time-Sharing System," *IBM J. Research and Development*, vol. 25, no. 5, 1981, pp. 483–90.
[4] R. Russell, "Virtio: Towards A De-Facto Standard for Virtual I/O Devices," *SIGOPS Op. Sys. Rev.*, vol. 42, July 2008, pp. 95–103.
[5] S. Soltesz *et al.*, "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," *SIGOPS Op. Sys. Rev.*, vol. 41, pp. 275–87, Mar. 2007.
[6] P. Padala *et al.*, "Performance Evaluation of Virtualization Technologies for Server Consolidation," HP Labs tech. rep., 2007.
[7] J. Liu, "Evaluating Standard-based Self-Virtualizing Devices: A Performance Study on 10 GBE NICS with SR-IOV Support," *2010 IEEE Int'l. Symp. Parallel & Distrib. Processing*, pp. 1–12.
[8] G. Wang and T.S.E. Ng, "The Impact of Virtualization on Network Performance of Amazon ec2 Data Center," *2010 Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[9] A. Iosup *et al.*, "Performance Analysis of Cloud Computing Services for Many Tasks Scientific Computing," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 22, no. 6, 2011, pp. 931–45.

[10] A. Gordon *et al.*, "Eli: Bare-Metal Performance for I/O Virtualization," *Architectural Support for Programming Languages & Op. Sys.*, 2012.

[11] R. Shea and J. Liu, "Understanding the Impact of Denial of Service Attacks on Virtual Machines," *2012 IEEE 20th Int'l. Wksp. Quality of Service,* 2012.

Biographies

RYAN SHEA (rws1@cs.sfu.ca) received his B.Sc. degree in computer science from Simon Fraser University, Burnaby, Canada, in 2010. He is currently a Ph.D. candidate in the Network Modeling laboratory at Simon Fraser University, where he also completed the Certificate in University Teaching and Learning. He has worked as a network administrator with various non-profit groups. His research interests are computer and network virtualization, and performance issues in cloud computing. Recently, he received the best student paper award at IEEE/ACM IWQoS'2012 for his paper "Understanding the Impact of Denial of Service Attacks on Virtual Machines."

JIANGCHUAN LIU [S'01, M'03, SM'08] (jcliu@cs.sfu.ca) received his B.Eng. degree (cum laude) from Tsinghua University, Beijing, China, in 1999, and his Ph.D. degree from the Hong Kong University of Science and Technology in 2003, both in computer science. He is a recipient of a Microsoft Research Fellowship (2000), a Hong Kong Young Scientist Award (2003), and a Canada NSERC DAS Award (2009). He is a co-recipient of the IEEE ComSoc Best Paper Award on Multimedia Communications (2009), Best Paper Award of IEEE GLOBECOM '11, and Best Student Paper Awards of IEEE/ACM IWQoS '08 and IWQoS '12. He is currently an associate professor in the School of Computing Science, Simon Fraser University, and was an assistant professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong from 2003 to 2004. His research interests include multimedia systems and networks, wireless ad hoc and sensor networks, and peer-to-peer and overlay networks. He is a member of Sigma Xi. He is an Associate Editor of *IEEE Transactions on Multimedia*, and an editor of *IEEE Communications Surveys and Tutorials*. He was TPC Vice Chair for Information Systems of IEEE INFOCOM '11.