A survey of network device virtualization

Debadatta Mishra

Abstract

Hosting multiple operating systems(OS) on a single physical hardware is very useful for server consolidation and optimizing resource utilization. There are various other benefits like de-coupling of OS from hardware, mobility of OS and applications due to virtualization. Several interesting and challenging problems due to virtualization come into surface. Virtualizing I/O devices is one such issue that has been the focus point of many researchers and computer scientists for a long time. Networking, being an essential part of any OS is an important aspect of I/O virtualization. In this report we study the challenges in virtualizing a Network Interface Card (NIC) and proposed solutions to overcome the issues. We also aim to find out the open problems in this area for further research and enhancement of network device virtualization.

1 Introduction

Virtualization allows multiple Operating Systems (OS) to run on a single hardware. A software layer known as Virtual Machine Monitor (VMM) or hypervisor is responsible for providing an execution environment for multiple guest machines or Virtual Machines(VMs). A computer consists of different hardware components like CPU, memory and I/O devices. Every hardware resource need to be virtualized(Figure 1) so that it can be presented as an isolated resource to each guest VM. The initial problems that virtualization had to solve was virtualizing CPU and memory because these two resources are the core of any computer system. However, any computing system is not complete without the input output (I/O) devices like mouse, key board, hard drive and network interface card (NIC) etc. Providing all these I/O interfaces to the guest VM is a must to make virtualization an useful technology. But the I/O devices are not part of the core architecture and are different from CPU and memory by the way they are designed and interfaced.

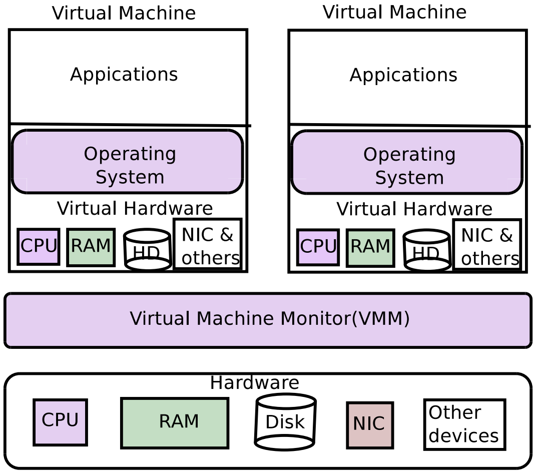


Figure 1: Virtualization of all hardware resources

1.1 How I/O devices are different?

Speed of I/O devices are order of magnitude slower than the CPU speed. Directly involving CPU for I/O operation in a synchronized manner wastes a lot of CPU cycles. Random Access Memory(RAM), even though faster than I/O devices causes performance degradation because the access of memory from CPU is done in a synchronous manner. Faster memory like cache memory is used in conjunction with RAM to improve the overall memory access speed. With I/O devices this approach is not yet adopted in any architecture. The speed mismatch is not the only reason for not adapting a synchronous I/O model. The other reason is the external dependency for an I/O event like receipt of a packet from the network. A packet can be received at any point of time making the I/O events aperiodic in nature. So periodic polling for packet arrival may not be a good solution. To handle these difficulties, the I/O event handling is designed to be asynchronous using interrupt mechanism.

Direct Memory Access (DMA) is a technique to offload data copy to and from the device to main memory from CPU to the DMA hardware. The DMA memory access does not happen via the memory management unit (MMU) of the processor. Thus the memory is addressed using the physical address as no address mapping mechanism like MMU is present in the DMA controller. For device sharing between multiple software entities those are isolated in memory by MMU, special DMA address mapping technique is required to enforce memory isolation.

Computer resources like processor and memory are by design sharable at a resource level. For a multitasking OS like Unix, many processes share the same CPU in a time sharing fashion. If we put CPU privilege levels and protection aside, at any point of time a process can be the owner of the whole CPU resource. At the next instant another process takes control of the CPU. This is possible because CPU state is transparent to the OS and can be saved and restored in software. Similarly virtual memory techniques like segmentation and paging support flexible partitioning of physical memory among different processes. I/O devices being peripherals and diversified, the OS is designed to work at an abstraction that is provided by the device drivers. Obviously the device drivers and device vendors don't take multitasking nature of the OS into their design and thus neither they provide partitioning of the I/O resource at the device level nor they provide a transparent save and restore of device state.

Providing only device state save and restore will not result in I/O partitioning unless the asynchronous nature of I/O events are handled differently. Consider a hypothetical I/O device providing transparent state save and restore and any software execution unit like a process has a corresponding device state that is loaded onto the device when it is scheduled on CPU. The external event occurrence during that period may not actually belong to the running process and thus the device must have capability of event classification and queuing based on the external event semantics. This not only makes the design of I/O device more complex and non-generic but also the device interfacing and interrupt mechanism becomes complex at the same scale.

Providing partitioned I/O at hardware level like memory partitioning is another approach for sharing the same device by many software entities. For an ideal partitioning, an I/O device may be dynamically divided into multiple devices such that each device becomes an independent device. But the I/O interfacing mechanism like interrupts and interconnects should also be partitioned for non-overlapping use of I/O resources. The OS should have the ultimate control to ensure efficient and fair I/O usage as in case of memory management.

1.2 I/O virtualization vs CPU and Memory virtualization

Sharing of processor between VMs is a logical extension of sharing it between processes in a multitasking OS. The main issue with CPU virtualization is to efficiently provide a mechanism of isolation between VMs. Most of the processor architectures provide different privilege levels of execution (Ring 0-3 in x86 for example). In a non- virtualized system, the OS runs in the highest privilege level and the user applications at the lowest. With virtualization, VMM is required to be executed at the highest privilege level. The guest OS now should be executing at a lower privilege level and should not have access to the CPU resources that can impact other VMs. Different software techniques like binary rewriting, paravirtualization and hardware extensions like Intel VT-X and AMD SVM are proposed and used today. Interested readers can refer [2], [3], [4] for more discussion on CPU virtualization.

Partitioning memory between processes using paging and segmentation can be extended to partition memory between VMs. This requires one more level of paging that would convert the guest VM notion of a physical page to a machine page. Software techniques like shadow paging and direct paging and hardware techniques like extended page tables (EPT) are proposed and used today. For more discussion on memory virtualization please refer [2], [3] and [20].

One important question with respect to I/O virtualization is whether we can use techniques for CPU and memory virtualization for I/O virtualization. To answer this question, we go back to Popek and Goldberg [1] formal requirements of an instruction set architecture (ISA) that has led to the existing software and hardware techniques for CPU and memory virtualization.

- The instruction abstraction of underlying hardware provided to the guest OS must be same as the original ISA. This property is commonly known as *equivalence* and is required if we want to run unmodified OS on top of the VMM. Any VMM that requires even small modifications to guest OS does not meet the equivalence property.

- Most of the guest OS operations should be carried out without the involvement of VMM. This is a qualitative requirement known as *efficiency* which rules out pure software emulators from the class of VMMs.

- Every resource in a computer is completely controlled and managed by the VMM. A guest OS should not use any resource that is not allocated to it and VMM can gain control of any resource at any point of time. This property is a mandatory requirement for a hypervisor to ensure isolation and security.

Considering the nature of the available I/O devices and their interfacing in a computer system, they need to be treated differently from resources like CPU or memory. We discuss the applicability of Popek and Goldberg requirements on I/O devices and throw some light on issues of achieving them as follows.

Equivalence for I/O virtualization can be thought as providing the same device interface to the guest VM. The peripheral nature of device relax this condition because most of the OS has device drivers as modules and not part of the core kernel design. Thus a changed device interface may require a separate device module in the guest kernel. However the abstraction at which the core kernel deals with the device driver must be the same as the physical device driver. This is easy to achieve as the design of the OS is such that same device from many vendors can be interfaced.

Efficiency for I/O devices would mean to perform device operations without VMM intervention. This is hard to achieve because of the inherent nature of currently available I/O devices as discussed above. If the VMM provides partitioned I/O devices in software, the overhead of VMM involvement is inevitable. Having device level resource partitioning may help to improve the efficiency but the VMM still need to provide proper management for resource assignment and CPU interfacing. Interrupt partitioning requires partitioning of CPU resource assignments like interrupt pins and interfacing hardware like programmable interrupt controller.

Resource control for software provided device abstractions is easier to achieve. If the VMM provides a virtual device to the guest and carry out operations on behalf of the guest VM, it can intervene any device operation by the guest. Thus isolation at different levels like virtual resource isolation, fault isolation and performance isolation can be achieved by VMM very easily. Static partitioning and assignment of the hardware resource at hardware level cease the power from the VMM to intervene in guest operation. The balance between autonomy of guest direct operation and VMM capability to intervene must be properly thought out for this kind of a design so that it can be both efficient and controlled.

1.3 Scope of the study

In this report we will focus on one of the I/O devices, the Network interface card (NIC) which enables communication between computers. The cloud provider creates a VM in its cloud and gives remote access to the end user. Thus, use of virtualization in cloud without network connectivity is difficult to imagine. Services like web server, mail server etc are accessed remotely over the network. For a VM that hosts such applications, network connectivity is inevitable. Network I/O is also different from other I/O devices because of the bandwidth requirements and the unpredictable nature of I/O activity.

Virtualization is used differently for people working at different levels of abstraction. End user wants to use the virtual machine(VM) the same way as any physical machine. The OS developer does not want to modify the OS and wants an equivalent hardware abstraction. The VMM designer wants to make the design more efficient in terms of simplicity, performance and scalability. We focus on I/O virtualization requirements taking example of the Network Interface Card (NIC). A single physical NIC need to be presented as multiple distinct virtual NICs which can be programmed and used by each guest VM using a network device driver software. We present the requirements from different user perspective for NIC virtualization in Section 2.

As in case of early days in processor and memory virtualization, I/O virtualization was mostly done in software. Only recently there are some advanced virtualization friendly device features. Majority of solutions for NIC virtualization are software only solutions. There are also some interesting solutions using the advanced NIC features. All these techniques suggest different architectures to address the challenges of I/O virtualization and meet requirements in their own way. We present broad categories of all proposed techniques for network device virtualization, evaluate each against the requirements and discuss about the proposed enhancements in section 3 and compare each technique in section 4.

Based on our analysis of different approaches for NIC virtualization, we would like to come up with future direction of research for efficient network virtualization. We would like to state the points of further investigation to make network virtualization better.

2 Requirements of network device virtualization

From a VM user perspective, the virtual NIC can be connected to some network and it can communicate to outer world through the NIC. The user only expects a similar activity like plugging in the network cable and expect networking to be up and running. From a device driver developer's perspective she should know about the detailed interfaces of the device. A hypervisor designer may want to make the network virtualization overheads less and ensure isolation in various axes like fault,reliability and fairness. We discuss each of the classes of requirement in the next subsections.

2.1 End user perspective

Below listed are some key aspects that an end user wants.

- The user can create virtual NICs by some software interface as it can insert a physical NIC into mother board. There may be a limitation on the maximum number of interfaces that a single VM can create and use.
- There should be some interconnect method provided just like the physical world (like a switch, bridge or another interface) which can be used to communicate.
- All network related functionality like browsing,firewalling, chating etc should be available to the user.

The above requirements necessitates a management component in a VMM that provides interfaces to create virtual NICs. Also some virtual interconnect method like a virtual switch or a software bridge need to be provided by the hypervisor in which the physical NICs are already part of. The third requirement is more trivial and mostly easily achieved because of inherent layered implementation of network stacks.

2.2 OS developer perspective

Ideally the OS providers do not want any change in the main kernel to support virtual network interface (VIF). Most of the OS provide a loadable module model for supporting different I/O devices. Keeping this in mind we list down the expectations of a OS provider or developer below.

- The device driver for the VIF if not already available can be written as a module using the existing OS API and helper routines.
- The virtual network device driver should interface with the network stack in the same way as any physical device driver. The VIF driver for example should not directly call transport layer receive routines after receiving a packet. If this is done, the device driver must implement complete IP layer processing and provide same IP interfaces as that of the particular OS to upper layers. This design makes the driver intrusive and complex. So, the driver should implement generic device independent interface for device operations that is used by the upper layers.
- The device driver should not drastically impact the OS assumptions about the resources. For example the device should not allocate memory that is far more than what a normal network device is supposed to allocate. If such an exceptional behavior is present, the OS will have to be modified or additional VMM intervention will be required.
- The device interfaces should be clearly specified so that a developer can write a driver for this. Mostly the device driver is developed by the VMM developers with the knowledge about the underlying hypervisor.
- Standard NIC parameters for efficiency (like NIC coalesce settings) need to be clearly stated. If any additional tunables are provided, their behaviors must be specified.

Ideally, providing a physical NIC equivalent interface would make the life of guest OS developer easy. However, this may result in violating the resource requirement expectations of the OS. Sending a packet through the VIF for example might take longer compared to a physical NIC or the maximum send rate of the physical device may never be achieved with multiple VMs sharing the link. Thus, the network device virtualization must consider all of the above for seamless VIF integration with the OS.

2.3 VMM designer perspective

The VMM is responsible for multiplexing the physical NIC as multiple virtual NICs. Following are the points to keep in mind for network device virtualization.

- The isolation requirements can be further classified into resource isolation, fault isolation and performance isolation. Resource isolation does not allow any other VM to access the VIF resources that is not assigned to it. Malfunction or crash of a VIF in any guest domain should not impact other VIF functioning as long as the physical interface is up and running. Performance isolation is required to ensure fairness in multiplexing of VIFs into a single NIC. One VIF should not dominate the use of NIC in presence of traffic from other VIFs.
- Most of the available physical NIC hardware should be virtualized by the VMM. The VMM might implement the drivers in the core hypervisor code or may adapt some other model like having a special guest domain hosting the drivers. In hosted virtualization or type II VMM, the VMM runs as part of a host OS and the host OS may provide the device drivers for various kinds of NICs.
- The VIF should be as efficient as the physical NIC. This is a qualitative requirement just like efficiency requirement of Popek & Godberg. But unlike CPU

emulation we can not rule out device emulation because of the difference between two resources as explained in section 1. We propose two metrics using which we can evaluate efficiency of a technique. Firstly, if only one VIF is using a physical NIC at any point of time, the maximum bandwidth available to the VIF should be the same as the physical NIC bandwidth limit. Secondly, the resource usage for achieving certain bandwidth in case of a VIF should be close to that of achieving same bandwidth using a physical NIC. Thus efficiency for network device can be measured in terms of extra resource usage while using the VIF compared to the native case.

- The VMM can intervene and modify the behavior of any device operation initiated by the guest on the VIF at any point of time. This gives a better resource control and keeps the guest VM independent of the hardware. One of the advantages of virtualization is a notion of *software computer* that can be moved and replicated like any other software. If the VMM can not intervene at any time to relinquish control of the hardware, the guest VM no more remains a software computer. The solution approach for which the granularity of resource control is coarse might result in difficulty in supporting features enabled because of virtualization.

Isolation is a must for any resource virtualization and NIC is no different. Most of the current designs try to trade-off between efficiency and resource control as we will see in the next section.

3 I/O virtualization techniques

A normal network device not supporting any in-device partitioning is virtualized in software by the VMM as shown in Figure 2 . Physical NIC that is connected to an external switch is initialized by the NIC device driver and owned by the VMM. The VMM provides multiple virtual interfaces to the guest VM. The guest OS initialize and use the VIFs by loading the driver for VIF. The VMM provides a software switch that routes the traffic between the physical NIC and the virtual interfaces.

If we want to present the guest VM a VIF that is exactly same as providing the the physical NIC or some other well known NIC, we have to provide all the control and data registers of that NIC so that the device driver for the physical NIC can be used. This method is known as device emulation. Equivalence is the biggest advantage of this method, but this method incurs a lot of overhead and efficiency is a big concern.

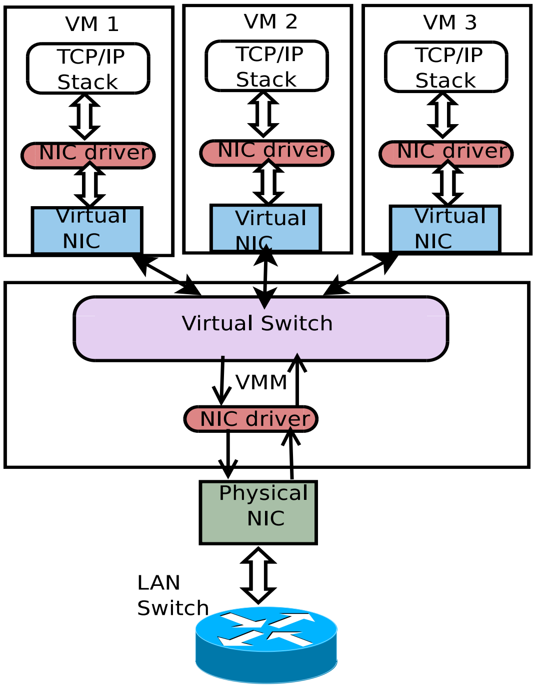


Figure 2: Virtualization of network interface

Thinking in terms of NIC features instead of NIC hardware design can be one more angle to NIC virtualization. In stead of providing the same hardware registers and interfaces like a physical NIC, a VIF with design specifications that support packet send and receive capability can be presented to the guest. Para-virtualization adopts this method to maintain equivalence at abstract device usage level rather than at device resource level. This is in the same line of thought for CPU and memory virtualization where a subset of the underlying hardware architecture and the instruction set is exposed to the guest to OS. Para-virtualization in general has the disadvantages of not meeting the equivalence property, thus requires change in the guest OS. But, if the CPU and memory can be virtualized using hardware extensions without requiring modification in the guest OS, providing NIC equivalence at feature level requires a special device driver module in the guest OS and does not require any other change in OS design. Most of today's software methods use this model for network virtualization.

Virtualization support in hardware has been a recent trend for efficient virtualization. Intel VT-x and AMD SVM are example hardware extensions to support efficient virtualization. These hardware extensions help to achieve memory isolation and processor virtualization by using hardware features. However they do not provide any support for efficient I/O virtualization. To address the challenges of I/O virtualization, there have been hardware extensions in the NIC design. Single root

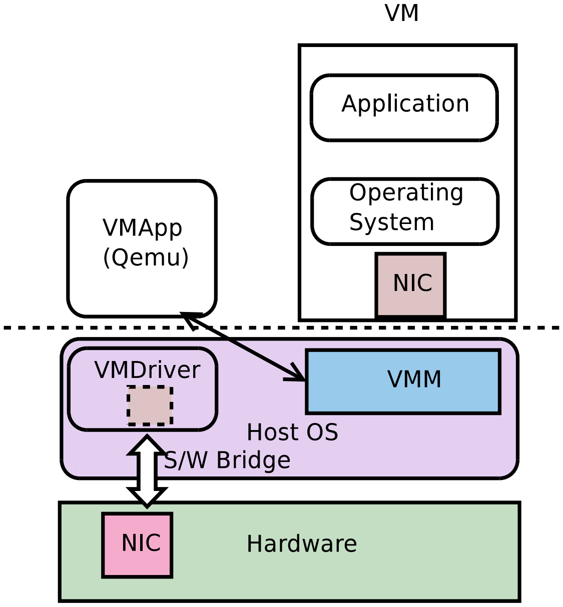


Figure 3: Emulated I/O in hosted virtual machine

I/O virtualization (SRIOV) with Intel VT-D or AMD IOMMU is an example of such a solution that we will discuss in this section.

3.1 I/O virtualization using device emulation

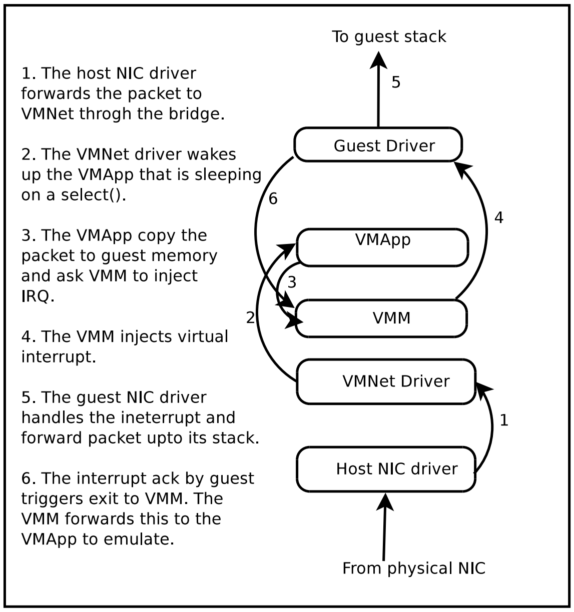
Device emulation is one of the initial approach for device virtualization. For providing a device interface same as the hardware device, the VMM provides an software emulated I/O device that has the exact same hardware structure like registers maintained in software. When the guest device driver tries to query and configure the device using I/O instructions, all these instructions cause a trap as I/O instructions can be executed in the most privileged level of CPU operation. The VMM handles these traps by executing equivalent operations on the device maintained in software. The trap and emulate model for network device operations originated by the guest VM driver can be implemented in software in a fairly straight forward manner because of the synchronous nature of traps. However, injection of external event as interrupt in VIF like a packet arrival can be an involved process because of asynchronous nature of interrupts.

3.1.1 Emulated network I/O architecture

The emulated I/O model of VMWare workstation [5], a Type II VMM is shown in Figure 3. The device emulator VMApp runs as an user space process in the host machine. The VMDriver provides a dummy device that works as an interface between the VMApp and VMM . In the given model, the CPU virtualization is handled completely by the VMM. As long as the guest VM is executing non-privileged instructions, the VMM does not intervene. If the guest VM performs some I/O operation like reading or writing to a I/O register, the VMM will intercept those operations and forward it to the VMApp to handle the intended operation in software. An updation of TX queue descriptors by the guest VM for example can result in a send() system call by the VMApp. The hardware interrupts also cause the VMM to take control of the CPU and re-inject the interrupt into the guest VM. VMDriver implements a software bridge between the guest NIC and the physical NIC to send and receive packets in a protocol transparent manner. For a NIC emulation, the guest OS is presented with a virtual NIC that has the exact same hardware interface as the physical NIC. The guest OS does all the initialization of the NIC using the same physical device driver software. The hardware transactions however are trapped and emulated by the VMM taking help of VMApp. KVM also adapts a similar device model using Qemu as the device emulator.

Sending of a packet from the guest VM is initialized by the driver setting up the transmit descriptors. This requires multiple device I/O instructions and each of them is trapped by the VMM and forwarded to the VMApp. For the I/O instruction that results in an actual send is handled by the VMApp by calling the write() system call on the VMNet device. The VMNet device puts the packet in the software bridge to send the packet through the physical NIC device. After the packet is transmitted by the physical NIC, an interrupt is asserted by the device causing VMM to forward the interrupt to the guest VM through the VMApp.

Packet receive (Figure 4) by the guest VM is more complicated than sending a packet. The VMApp blocks on select system call for the VMDriver device. When a packet arrives in the physical NIC the host driver handles the interrupt and pass the packet up in the host network stack. If the packet is destined to the guest VM, the bridge forwards the packet to the VMDriver. The VMDriver wakes up the blocking VMApp by passing the packet to it. The VMApp copies the packet onto guest memory and requests VMM to raise a virtual interrupt for the guest. The guest NIC driver handles that interrupt and forwards the packet for further processing in the guest stack. In most device interrupt semantics, the interrupt handler must acknowledge the interrupt by setting some bits in hardware register. When the guest device driver does that, the VMM intercepts and forwards the I/O command to the VMApp to emulate.


1. The host NIC driver forwards the packet to VMNet throgh the bridge.

2. The VMNet driver wakes up the VMApp that is sleeping on a select().

3. The VMApp copy the packet to guest memory and ask VMM to inject IRQ.

4. The VMM injects virtual interrupt.

5. The guest NIC driver handles the ineterrupt and forward packet upto its stack.

6. The interrupt ack by guest triggers exit to VMM. The VMM forwards this to the VMApp to emulate.

Figure 4: Packet receive flow in emulated I/O

3.1.2 Evaluation of requirements

Positives: The end user requirements and OS developer requirements are met in an ideal manner in emulated network device virtualization.

- The guest OS need not be modified at all if it contains driver for a well known network card that is provided by the VMM. The VM user need to create a bridge consisting of the physical NIC and the virtual NIC.

- As the whole device is implemented in the guest OS, there is no requirement of abstract device independent layer from the OS. Supporting NICs from different hardware vendors is offloaded to the host OS by virtue of a hosted architecture.

- The isolation and resource control aspects are taken care by the VMM interception at each I/O instruction for hardware access. The VMM can take charge of the resource at the granularity of each network I/O operation, thus enforcing any policy required for guaranteed isolation

Issues: Efficiency and performance aspects of emulated network I/O is unacceptable in virtualization systems. The initial design could only drive 20% [5] of the send bandwidth of the physical NIC.The major performance degradation is because of VMM intervention for each I/O instruction. In a hosted VMM scenario, the guest VM runs as a process on the host OS. For every I/O instruction emulation by the VMApp, there is a host level process context switch. For packet receive there is more than one context switch resulting in more overhead. The packet copy in receive path is one more unavoidable additional operation that requires additional CPU resources.

3.1.3 Performance improvements

All I/O instructions do not result in actual I/O operation. For example, several I/O instructions are executed by the guest driver before actually triggering a packet transmit. Context switch from the guest mode to the host mode application (VMApp) for instructions that do not result in any meaningful activity (like actually making a write system call) is not necessary. The VMM can directly apply these modifications to the virtual NIC without a context switch.

Combining multiple packets at the VMM before calling VMApp transmit routine helps in reducing the context switch overheads and system call overheads. The VMM should make sure that the send combining does not introduce a lot of delay in packet transmit. Carefully designing aggregation parameters can make this optimization useful without impacting the transmit latency.

Shared memory between VMDriver and VMApp can be used as an alternate to blocking select mechanism for packet receive notification. The VMApp blocks on select() call to get a notification when a packet arrives. This method incurs system call overhead in receive processing. VMApp can poll the shared memory between VMApp and VMDriver to determine arrival of a packet.

Even after applying all the above optimization, the transmit takes four times more CPU [5] then the native case. The receive overheads is more than send overheads because of the extra packet copy and context switches.

3.2 Split driver para-virtualized model

A physical NIC has several registers and a complex hardware state. This complexity is necessary and useful for a physical NIC interfacing with a mother board. If the equivalence requirement can be compromised, providing NIC with capability to transmit and receive to a VM does not hamper any other requirement. So if a minimal NIC equivalent device is presented to the guest for packet send and receive[6], not only the network virtualization becomes simpler, it also can become very efficient in terms of performance and suppotability. Xen uses a special guest called domain-0 or the control domain that runs a standard OS like linux and has drivers in-built for almost all types of available NICs. The domain-0 acts as a management and control interface for the end user and handles the physical device driver operations. The domain-0 also holds the backend of the virtual interface and provide a mechanism for guest VMs to carry out NIC operations through it. This model proposed by the Xen is currently adopted in some form or other by most of the open source and commercial hypervisors.

3.2.1 Xen infrastructure for PV device model

Xen provides a lot of virtualization primitives to make the implementation of split device drivers simple. We discuss some of the relevant technique in the following paragraphs.

Hypercall is a way to carry out privileged tasks from a guest domain which is running in lower processor privilege level. It is like a system call in normal OS using which a user process enters the kernel mode. There are at least 48 hypercalls provided by Xen for different class of privileged operation. The driver domain or domain-0 is allowed to execute all the hypercalls while the normal VMs are only allowed a subset of it.

Xenstore is a in-memory storage system shared between all guest VMs. Along with many other information about VMs, it contains all the information about the available devices and device parameters for each domain.

Event channel is a virtual counterpart of physical interrupts and provides asynchronous event delivery framework. A event channel can be bound to a physical interrupt, virtual interrupt (VIRQ) or another event channel. Domain-0 creates event channels for each of the physical device except for the timer device and makes a hypercall to bind the physical IRQ to the event channel so that it gets an interrupt when a physical interrupt is asserted by the device. Inter-domain event channels provide a two way communication method between a pair of domains (mostly domain-0 and any other VM) and required for split device model as we will see in the next subsection.

I/O channel or I/O rings is a shared ring producer consumer data structure used to enable a bi-directional request response exchange between two domains. For device communication I/O ring is created by non-privileged domain and shared with the domain-0.

Grant mechanism is a method of allowing another guest to use some memory pages temporarily. This is useful for passing a large amount of data from one domain to another in a secure way. The grant sharing is exclusive i,e at any point of time only one domain can use the page. Granting page,accepting a granted page and revoking a grant are done through the hypervisor for security and isolation.

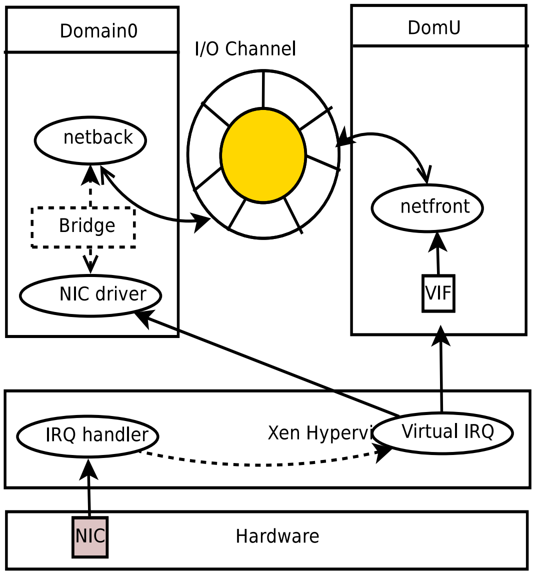


Figure 5: Xen split network device model

3.2.2 Xen network I/O architecture

The split device model is shown in Figure 5 . The domain-0 physical device driver is responsible for sending and receiving network packets from the wire. The guest VM configured with virtual interface (VIF) runs the frontend driver called netfront and the domain-0 instantiate backend driver instance called netback. The VIF is bridged with the physical interface using a software bridge provided by the OS. Thus a packet received by the physical drivers goes through the bridge to the correct VIF. An I/O channel or I/O ring is created between frontend and backend drivers for co-coordinating data transfer between frontend and backend. The I/O ring is a shared memory created by the guest VM and mapped to domain-0 through the hypervisor. The backend and frontend use it as a mechanism of placing requests and responses to each other. Virtual IRQ is an equivalent mechanism of physical IRQ to notify the guest about receipt or transmission completion of a network packet. Every packet flowing between the guest VM and physical NIC goes through the I/O channel notification and software bridge.

Transmission of a packet (Figure 6) from the guest protocol stack is initiated by calling generic network device transmit routines. The netfront driver puts the transmit descriptor into the I/O ring. The transmit descriptor as the name suggest contains information of the network buffer holding the packet. The backend driver is notified by using a hypercall to the Xen hypervisor and the hypervisor triggers a VIRQ to the domain-0. The net-

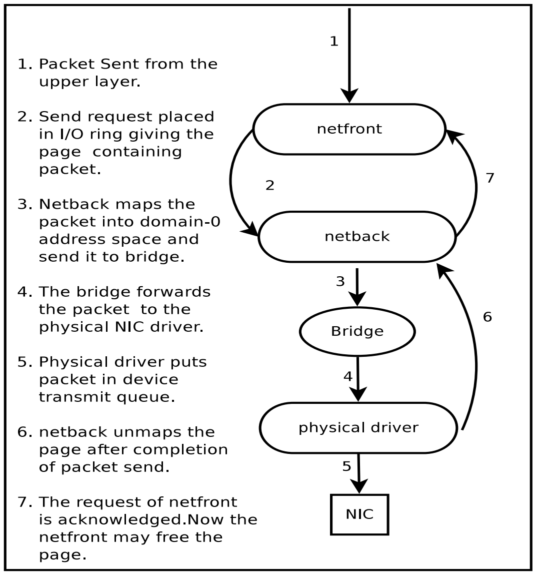

1. Packet Sent from the upper layer.
2. Send request placed in I/O ring giving the page containing packet.
3. Netback maps the packet into domain-0 address space and send it to bridge.
4. The bridge forwards the packet to the physical NIC driver.
5. Physical driver puts packet in device transmit queue.
6. netback unmaps the page after completion of packet send.
7. The request of netfront is acknowledged.Now the netfront may free the page.
1
netfront
2
7
netback
3
6
Bridge
4
physical driver
5
NIC

Figure 6: Network packet transmit flow in Xen para-virtualized driver

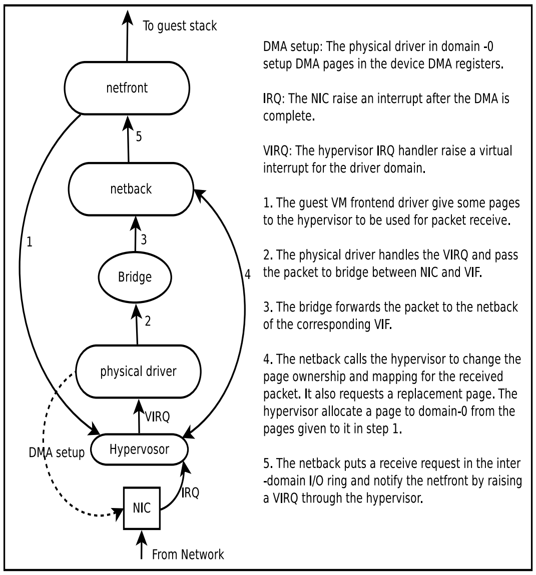

To guest stack
netfront
5
netback
1
3
Bridge
4
2
physical driver
VIRQ
DMA setup
Hypervosor
IRQ
NIC
From Network
DMA setup: The physical driver in domain -0 setup DMA pages in the device DMA registers.
IRQ: The NIC raise an interrupt after the DMA is complete.
VIRQ: The hypervisor IRQ handler raise a virtual interrupt for the driver domain.
1. The guest VM frontend driver give some pages to the hypervisor to be used for packet receive.
2. The physical driver handles the VIRQ and pass the packet to bridge between NIC and VIF.
3. The bridge forwards the packet to the netback of the corresponding VIF.
4. The netback calls the hypervisor to change the page ownership and mapping for the received packet. It also requests a replacement page. The hypervisor allocate a page to domain-0 from the pages given to it in step 1.
5. The netback puts a receive request in the inter -domain I/O ring and notify the netfront by raising a VIRQ through the hypervisor.

Figure 7: Network packet receive flow in Xen para-virtualized driver

back driver in domain-0 sends the network buffer page to the hypervisor to verify and map it to domain-0 address space. This is required to verify that the page actually belong to the frontend of the guest VM. The netback forwards the packet to the domain-0 stack where it flows through the VIF and NIC bridge to reach the physical NIC driver. Once the driver push the packet into the physical NIC and transmission completes, a call back is triggered to let the netback know that the packet is transmitted. The domain-0 unmaps the network buffer page and write a response to the I/O ring. The domain-0 sends an VIRQ to the netfront driver through the hypervisor to notify the completion of send. The netfront read the response and takes back the page for other use. There are some implementation optimization for notification mechanism. Instead of sending a notification through a VIRQ for every request or response, the sender notify the other end if it is not notified recently. Note that for every packet transmit there are two domain-0 page table changes by virtue of mapping and unmapping the network buffer.

The netfront driver releases some pages for packet receive to hypervisor before the actual receive. After the packet is received by physical NIC onto the DMA location specified by the domain-0 (Figure 7), an interrupt is asserted from the device. The hypervisor handles the interrupt and forwards it as a VIRQ to domain-0. The physical device driver in domain-0 does the normal NIC processing and sends the packet up on the stack. The software bridge forwards the packet to the corresponding target VF or the domain-0 IP layer depending on the destination MAC address. The target VM is made the owner of the page containing the network packet by netback through a hypercall. The domain-0 driver also requests a replacement page from the hypervisor to keep its own memory allocations constant. The hypervisor maps a page that was previously released by the guest VM to the domain-0 address space. After all the mapping/unmapping is complete, the domain-0 puts a packet receive notification in the I/O ring and raise a VIRQ. The netfront driver in guest domain receives the packet and pass it up to its stack for upper layer processing. Note that the packet receive does not require any data copy as the page flipping is used for page ownership changes. The possible overheads of this scheme are the mapping changes required for page ownership changes and packet classification in software bridge.

3.2.3 Evaluation of requirements

End user requirements: The user need to create a VIF and tag it to the bridge provided by the domain-0. The high level send and receive functionality is not altered, thus no network application is impacted.

OS developer requirements: The OS provider need to include a special device driver for the frontend. Xen being a para-virtualized hypervisor, change in guest domain is not only required for the network virtualization, thus the network model does not make equivalence worse.

Even for unmodified guest OS, the device driver can be loaded as a module provided all required virtualization primitives provided by hypervisor can be accessed and used. The latest versions of Xen that takes advantage of Intel VT-X to provide unmodified guest OS use split device drivers.

VMM requirements:

- The driver domain model proposed by Xen takes care of the supportability requirement very nicely as the domain-0 is a general purpose OS and has drivers in-built for almost all available physical device types.
- Resource isolation and fault isolation is guaranteed by this model for guest VM by the introduction of hypercalls to ensure safety. But if the domain-0 crash because of the physical device driver bugs, the whole system goes down. There are proposals of having a separate driver domain for hosting devices and a control domain for VM management to make fault handling better, but this is not generally used in Xen implementations.
- The control points for the network device in paravirtualized model is fine grained and hypervisor can intervene in every packet send and receive operation. For example implementation of high availability as proposed in remus[7], transmitted packets can be queued in domain-0 without sending them through the physical NIC and can be sent at a latter point. In Live migration, the device state can be recreated and exposed to the guest easily as the VIF state is in software and independent on the physical device state.
- The overhead of para-virtualized network device was high in the initial architecture[8]. For sending a packet using the VIF, the CPU cycles spend is close to four times more than the native NIC case while for a packet receive the overhead is nearly 250%. In the next subsection we present different improvements of the Xen paravirtualized network device architecture.

3.2.4 Performance improvements in packet transmission

Sending a packet requires two mapping changes, some hypercalls and one bridge forwarding. If multiple packets are combined into one large packet, then the per packet overhead will be largely reduced. To complement this approach there are some hardware features already used in native environment. In a typical physical network interface, apart from normal NIC features there are some additional optimization features to offload some protocol specific processing from the processor to the NIC. We highlight three such features of NIC that is useful for transmit side processing optimization. *Checksum offload* allows the protocol stack to offload the checksum computation for header and data to the NIC. *TCP segmentation offload (TSO)* is used to transmit a large volume of data (more than the TCP segment size) in one transfer operation. *Scatter gather DMA (SG)* allows one DMA read and combine operation from different physical addresses.Using SG, the software may provide different locations and lengths for protocol headers and data fragments.

The NIC having TSO capability fragments the large data into transmittable chunks, appends TCP protocol headers and sends it to the network. The initial Xen virtual interface did not support any of these features, thus checksum computation and segmentation was done in software causing large overheads. The TSO feature can give a large gain as per packet overhead is reduced by a big margin. However supporting this feature in VIF requires some additional issues to be taken into consideration. The diversity of different physical NIC features need to addressed. Also if the feature is not present in some NIC, it should be implemented in software.

A generic offload layer is proposed[9] that abstracts the NIC features and can be used by the split device model to offload NIC features. The offload layer configures these features in physical NIC if they are present in the device else it implements the features in software. This technique improves the send performance even if the NIC features are not present. The normal packet size in Ethernet (MTU) is 1500 bytes, however each packet is placed on a page of typical size of 4K. Thus sending large packets from the guest domain reduce the number of page map/unmap operations by a factor more than two.

For transmission of a guest packet in the current scheme, the page is mapped to domain-0 address space. This is not required for packets not destined to the domain-0 as the bridge forwarding only requires headers for deciding the target interface. To avoid page map and unmap, the headers are copied along with the send request onto the I/O ring. The domain-0 maps the page only if it is the destination of the packet, otherwise the page number, offset and length information is passed in the packet buffer to the physical device via the bridge. SG feature described above is needed to realize this optimization. The generic offload layer implements SG functionality in software if the feature is not supported by physical device.

3.2.5 Receive side performance improvements

The packet receive overheads can be attributed to the following.

- The driver domain provides the DMA pages to the NIC for data receive. The received data has to be made available at the guest VIF in some way. The page ownership flipping mechanism discussed above requires several page map and unmap operations. There is additional overhead of scrubbing pages in domain-0 before assigning it to guest domain for security reasons.

- The packet received by domain-0 need to be classified based on the destination MAC address by the software bridge. Software bridge implementations in OS consume some CPU cycles per packet.

- The domain-0 and guest domain may run in different physical processors. The caching benefits of processing the same packet is nullified because of this. Also context switches between domains might result in TLB flushes causing performance degradation. It is shown in [8] that the cache miss overheads are huge compared to the native case.

- For every packet received there are hypervisor involvements (VM Exits) for handling physical interrupt, device DMA setup, VIRQ injection and notifications. VM Entry and VM Exit by themselves require context save and restore apart from the Exit handler overheads.

Packet copy from domain-0 to guest VM: Grant mechanism introduced earlier can be used as an alternative method for sending the packet data from domain-0 to the target guest domain. The guest VM makes a grant hypercall to the Xen hypervisor with a set of pages along with the domain with which these pages to be shared. The hypervisor maintains a table for keeping information about the grant pages for all domains. The netfront driver sends the page numbers to the netback driver through the I/O ring.On receipt of a packet, the domain-0 backend driver calls a grant copy hypercall to the hypervisor providing the source page, destination page, destination VM and length of data to copy the packet. The hypervisor verifies the page ownership and copies the data to the target page. The guest domain after getting notification about packet receive, revokes the page grant and use the data. The above method is better as page mapping and unmapping results in TLB flushes, but introduces an additional copy operation. There are architecture specific optimization for copy by aligning data to certain byte boundaries. For example, Intel architecture suggest that if data is 64byte aligned, the data copy is more efficient.

Receive combining: *Interrupt coalescing* is a hardware technique to enable batching of interrupt delivery. A physical NIC having interrupt coalescing support can be configured such that it will not generate an interrupt in every packet receipt, rather it will generate interrupt after certain number of packets are received or at some minimum time interval, which ever is earlier. This technique can be applied to VIFs to reduce the number of virtual interrupts reducing the overheads[10].The frontend driver may configure a coalescing parameter in the VIF and the netback will trigger a notification accordingly. *Large Receive offload (LRO)* is a receive batching software technique that combines received packets for a particular target and delivers a single large packet to the upper layer. If LRO is implemented in domain-0, the number of VIRQ and channel notifications in guest domain will be reduced. However having coalescing at different levels may adversely impact the latency and specially affect traffic bursts. Thus, different coalescing schemes must be combined together to come up with the coalescing parameters to achieve best possible results without impacting latency.

Caching improvements: Different cache miss overheads in split driver model receive are very high as shown in Figure 4, 5 and 6 of[8]. This is primarily because of the lack of proper support of cache virtualization in Xen[9]and some other implementation issues that results in more cache pollution.

Super page mapping support in Intel x86 processor allow a single mapping entry in page table from a set of contiguous virtual address range to a set of physically contiguous pages. This reduces the number of entries in page table and increases the TLB efficiency. Initial Xen implementations did not allow guests to have super pages, thus the TLB miss was more in this case. Support for huge pages in guest reduce the granularity of memory assignment and protection to a guest (i,e one page or 4K). There are also issues with other features related to memory virtualization like content based sharing and ballooning.

Global page table mapping allows certain page table entries to be marked as global and they are not purged across TLB flushes. In native OS design, the kernel page table entries are made global for making kernel code execution faster by ensuring TLB hit for a kernel address access. The Xen hypervisor marks its own pages as global so that the TLB entries remain intact across domain switches. The guest kernel pages are not marked

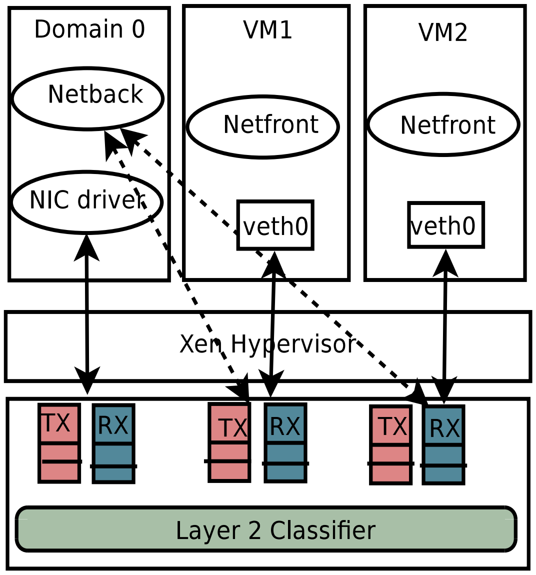


Figure 8: RX/TX queue assignment in multi-queue network device

as global as global page table feature is not exposed to the guest VM. Exposing global page table mapping to guest requires the complete TLB flush in domain switch. If a guest is mostly scheduled on a CPU and there are very few domain switch, this can help in improving the TLB hit ratio.

Multiple transmit and receive queue in NIC: The challenges of network I/O virtualization and high speed network requirements with multiprocessor machines has forced some change in NIC hardware. Multiple queue NICs (Figure 8) provide multiple receive (RX) and transmit(TX) queues as opposed to single RX and TX queue in older NICs. The classification of packet based on destination MAC address or VLAN ID can be done in hardware.

There are proposals [12],[13] of making Xen split driver model more efficient by assigning different RX queues to different VM and doing a direct DMA onto the guest physical address.There is however a subtle isolation issue here as the DMA address must belong to the guest VM and this must be ensured by the hypervisor. There are hardware techniques like IOMMU to take care of this issue as we will see in the next section when we look into hardware assisted approaches. The isolation can be achieved in software using the page grant mechanism. The guest VM releases some grant pages to the domain-0 and the domain-0 sets up the device DMA queue choosing one of the freely available TX and RX pair. The domain-0 also sets up the hardware classification based on the MAC address so that the packet received by the physical NIC is put directly into the proper device queue. This method overcomes the overheads of software packet classification and packet copy. As shown in figure the domain-0 still has administrative control of the device queues and can relinquish the resources at any point of time.

3.3 Hardware assisted network virtualization

We have seen a glimpse of new hardware features enhancement to make network virtualization fast in the last subsection. A logical extension to multi-queue network device would be making a partition of NIC resources into multiple virtual NIC in the device itself. There are certain interconnect challenges associated with making more than one uniquely identifiable device functions present in a single end point of any bus interconnect method. Assigning a hardware virtual NIC directly creates one more problem for memory isolation. A malicious guest can give arbitrary address that is not assigned to the guest for DMA as there is no way of verifying whether that address belong to the guest. To ensure memory isolation, we need a memory management unit (MMU) for I/O devices that will transfer the guest physical address into machine address.

3.3.1 Hardware enhancements[14]

Most commonly used interconnect method in modern computers is Peripheral Component Interconnect(PCI). The PCI bus is a tree like structure in which every leaf element is a device. The root node of a PCI bus tree is called the PCI root complex. Every device connected through a PCI bus has a unique identifier that consists of the path information from the PCI root complex to the device. The PCI interfacing method provides device independent access method for different type of devices. A device endpoint is called a device function and is unique for each device. Supporting a device endpoint with multiple device functions requires additional support for identification and addressing. The PCI enhancements propose physical function device (PF) and virtual function device (VF) to distinguish between the main device and the logical devices present within the device.An advanced NIC provides multiple virtual function devices and those are identified and accessed as different devices using the PCI enhancement standardized by PCI-SIG community. Thus every PF and VF will have separate and isolated PCI memory using which their respective device driver software can communicate and manage the device end points.

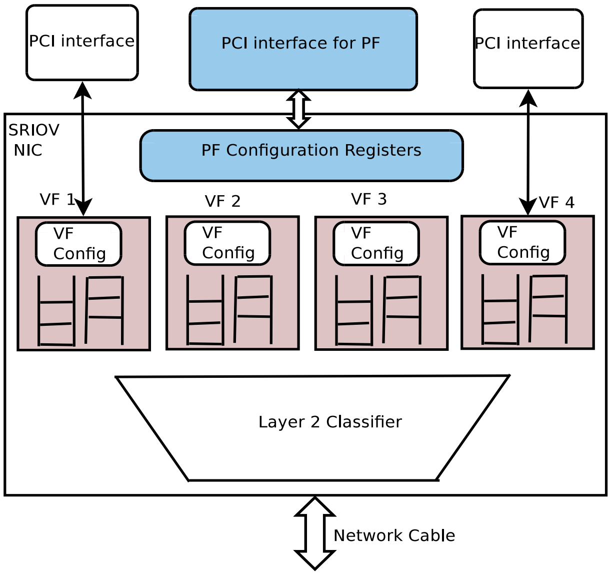


Figure 9: Logical design of a SRIOV NIC

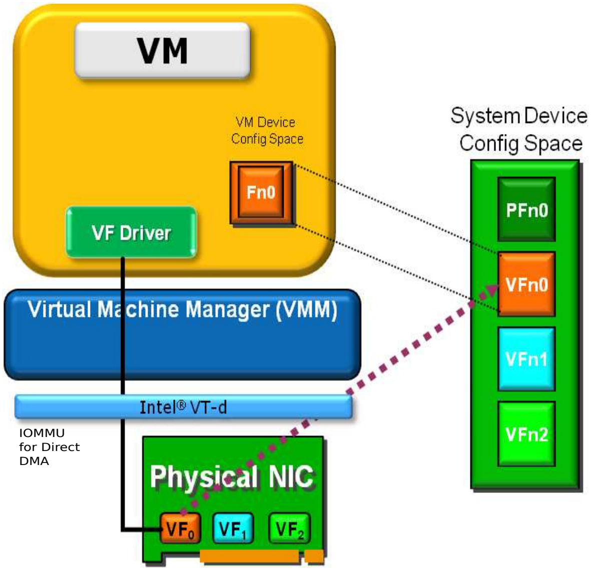


Figure 10: Assignment of VF to a guest VM. Taken from [14]

DMA address for any device is specified with a physical address as the access to the memory location happens through the DMA controller bypassing the MMU. Giving direct access of VF to a guest VM gives control of DMA setup for the device to the VM. There is no way to check the DMA address given to the VF by the guest VM. AMD IOMMU and Intel VT-d are the hardware techniques in the chip-set to provide a MMU for I/O devices. With IOMMU, before the DMA controller starts doing DMA into the guest VM provided memory location, the guest physical address is mapped to machine address using the IOMMU mapping table. The IOMMU table mapping is maintained by the hypervisor and can not be accessed from a guest VM directly. Having a IOMMU for safety gives birth to another problem of repeated memory access to get the machine address from the physical address as in case of a MMU without TLB. A IO-TLB is a TLB equivalent hardware caching solution to make the address translation fast.

3.3.2 Single Root I/O virtualization(SRIOV)

The SRIOV NIC shown in Figure 9 consists of multiple transmit and receive queues as in case of a multiqueue device. There is also a packet classifier in hardware to demultiplex packets based on MAC address or VLAN ID. The main feature of a SRIOV card is to provide a logical NIC that consists of isolated resources.Each of such logical NIC in hardware is called as a virtual function(VF) and has an unique PCI identification and configuration space. The physical function (PF) has control over the whole physical NIC while the VF has an isolated access to the VF configuration registers. The VMM or the control domain during boot finds out the PF device and loads the appropriate driver. The PF driver initializes the VFs along with normal NIC device initialization. At this step the physical NIC is partitioned into multiple NICs in hardware and every VF has a unique PCI identification along with transmit and receive queues. The number of transmit and receive resources per VF can be configured through the PF driver.

In Figure 10 intel SRIOV card guest VF assignment is shown.The System device configure space can be the domain-0 for a Xen like hypervisor or the VMM itself for VMMs those hosts device drivers within themselves like VMWare ESX or in the host OS for a type II VMM like KVM. The VF device assigned to a guest VM appear as a normal PCI device in the guest device configuration space. This is done either by emulating the PCI infrastructure in the VMM or providing para-virtualized PCI interconnect method. In fully virtualized VMMs, a device emulator like Qemu is used for providing device infrastructure. The VF driver is a special driver because the access and configuration is not same as a physical NIC. The direct data copy from the device to the guest memory is achieved using the Intel VT-d direct DMA technology. A mailbox communication infrastructure between the VF and PF is provided in hardware for carrying out operations by VF driver that require access to registers outside the VF configuration space. There is a loop-back like channel between the VFs for transmitting

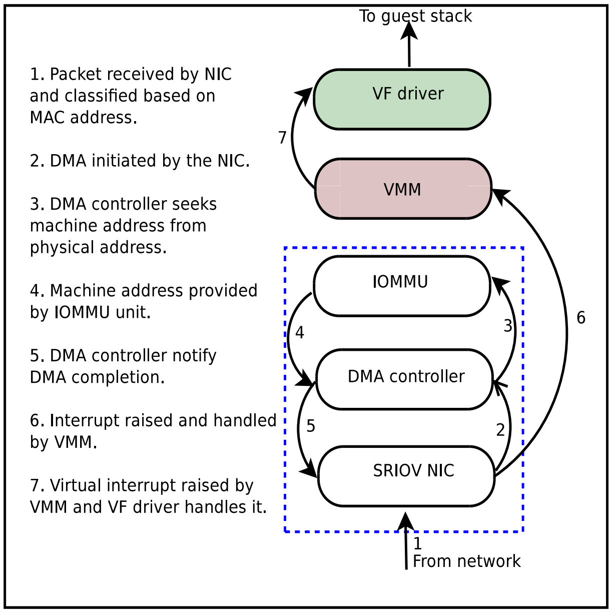

To guest stack
1. Packet received by NIC and classified based on MAC address.
2. DMA initiated by the NIC.
3. DMA controller seeks machine address from physical address.
4. Machine address provided by IOMMU unit.
5. DMA controller notify DMA completion.
6. Interrupt raised and handled by VMM.
7. Virtual interrupt raised by VMM and VF driver handles it.
VF driver
VMM
IOMMU
DMA controller
SRIOV NIC
7
6
4
3
5
2
1
From network

Figure 11: Guest receipt of a packet in SRIOV

traffic from one VF to the other. However there is a single interrupt line from the device to the CPU and all the VFs share this interrupt line for their operation. Thus interrupt de-multiplexing mechanism should be provided by the VMM to support multiple VF devices.

During the VF driver initialization the TX and RX queues are setup using the guest physical address. When a packet is transmitted, the transmit descriptors of the VF TX queue are modified by the VF driver in the guest domain to let the device know that there are outstanding packets to be transmitted. The device initiates a DMA from the guest provided guest physical address. The DMA controller get the corresponding machine address using the IOMMU mapping table provided by the chip-set. On DMA completion, the device starts transmission through the classifier.There are certain security and bandwidth control checks facilitated by the SRIOV NIC. The device checks for MAC address spoofing to ensure that the MAC address given to the VF is same as the originating packet MAC address.Rate control if configured for a VF can be applied to control the rate and drop packets if a VF is exceeding the configured limit.

When a packet is received from the network (Figure 11), the Layer 2 classifier puts the packet into the appropriate receive queue. The device starts the DMA into the specified location in the RX queue. The DMA controller requests address translation by the chip-set from the guest physical address to the machine address. Once the data copy to the guest address is complete, an interrupt is asserted by the device. The hypervisor handles the interrupt and re-routes it to the proper guest VM using virtual IRQ mechanism discussed in the last section. The guest VM handles the interrupt and continue processing of received packet.

3.3.3 Evaluation of requirements

End user requirements: From the end user perspective, the SRIOV method is little complicated for creating new a virtual interface if there are no intermediate VMM management interface managing the VF in a transparent manner. The VF to guest assignment should be automatic and seamless. A management module can be created in VMM to provide a transparent assignment of guest virtual interface to the VF. Other user requirements are better met in this case because the high level features of VF are same as any physical NIC.

OS developer requirements: The virtual function device presented to the guest is not a full fledged NIC device and need special device driver for access. From guest OS developer perspective VF device can be considered as one more device type and a separate driver module can be developed for managing the virtual function device. Intel provides its own driver for virtual function devices for its 1Gbps and 10Gbps SRIOV network cards.

VMM requirements: The design of SRIOV card takes care of most of the efficiency worries of the VMM designer.The packet classification is completely taken care of in the hardware. IOMMU direct DMA into guest address space takes care of the copy overheads in software virtualization techniques. IOMMU however can be costly for smaller packet sizes because of mapping overheads[19]. The interrupt routing overhead and PCI emulation overhead still remain and can not be removed using current hardware enhancements. The interrupt routing overhead in case of para-virtualized I/O might be smaller than fully virtualized SRIOV I/O, but there are no performance studies to validate this hypothesis.

The VFs are completely isolated in hardware such that no VF driver can access other VF resources and fault in one VF driver doesn't propagate to other. Performance isolation however is still an issue even if the NIC provides basic bandwidth control mechanism. This issue arises because the number of virtual functions available in a NIC is limited and if more number of virtual NICs are needed, the software method for providing VIF is used. A mechanism of resource allocation between the software multiplexing point and direct assignment need to be designed such that the fairness of resource allocation is achieved.

The SRIOV NIC hardware are standardized and can be implemented in a generic way if the PF and VF drivers are available by the device provider. But hardware assisted virtualization support like this does not cover the whole range of available hardware. So even if the SRIOV is a very efficient solution for device virtualization, this does not make the software based solutions redundant.

Assigning a VF directly to the guest VM gives complete control of the resource to the guest. This makes the granularity of resource control very coarse and impacts functionality like VM live migration.The guest OS no more remains independent of the underlying hardware and requires same hardware presence at the target for correct resumption of the VM. This problem arises because once configured, the hypervisor can not take back the resource from the guest in a transparent manner without abruptly ceasing the device from the guest. The granularity of control therefore is at the level of assigning VF or taking back the VF.

3.3.4 Performance improvements

The performance of hardware assisted virtualization is the best selling point of the technology [15], [16] . The better performance is a result of implementing classification and packet copy functionality in hardware. Intel SRIOV card can achieve near native performance with 50% more CPU. The increased CPU requirement is a result of VMexits due to interrupt and PCI virtualization. IOMMU also gives certain overheads for small packet sizes even though it is implemented in hardware because of repeated memory access for address translation. Reducing the number of interrupts by interrupt coalescing can help to reduce per interrupt processing overhead.The coalescing setting in both physical function driver and virtual function driver must be taken into consideration for proper coalesce setting so that the delay is not too high.Having a IOTLB for caching address translations help improve the IOMMU overheads.

Interrupt delivery without hypervisor interception is the ideal method for achieving best possible performance. Exit Less Interrupt (ELI) [11] is a technique to provide an almost direct interrupt delivery by making a simple assumption of temporal correlation of interrupt occurrence to the running guest VM. The CPU exceptions or traps can be deterministically handled by the running guest as explained in section 2. To handle interrupts bypassing the VMM, the VMM maintains a shadow interrupt descriptor table (IDT) for each guest VCPU provided by the VM and verified by the VMM.The VMM loads this shadow IDT onto the physical CPU register when a VCPU to physical CPU binding change. This method has OS implementation dependence and has a security issue as a malicious guest might handle interrupts that is meant for some other guest VM.

3.3.5 Resource control issues and solutions

The dependence of guest VM on the hardware negates the advantages of virtualization to some extent. Using a SRIOV NIC makes the VM heavily dependent on hardware and it becomes hard to achieve the OS and hardware independence. The issue of dependence can be visualized in two different angles. 1. If the source and target hardware have the hardware capability, is it possible to move a VM? 2. If SRIOV is available only at the source or the destination, can we move a VM?.

The first question is addressed partially by some proposed techniques. A VF state unlike a CPU state can not be saved and restored using software methods. Optimized device operation recording at the source VF driver [17] and replaying the device operations at the target to reach the same device state is one proposed solution for SRIOV NIC. There are some external events like a packet receive which can not be recreated at the target VM efficiently. Mostly these kind of operation have a side effect on read-only device states and can be emulated by the hypervisor. The number of dropped packets for example is a read-only register in a NIC and every access to the register can be emulated by the hypervisor. But there can be some hidden state change in the device for an external event for which either the dummy external events need to be created and replayed at the target or some device specific hacks are performed at the target to reach the state. The RX queue descriptor for example has a producer pointer that is moved by the device after receiving a packet and can not be changed by the driver software. Thus the DMA locations need re-adjustment for ensuring correct device operation at target.

The second problem of having virtual devices irrespective of available hardware features is a tough problem to handle. This can be argued to be a similar problem of running VM on different hardware architectures. Assuming same CPU architecture, still the problem of transparent device operations remain challenging.

Hardware extensions [18] to solve the problems of hardware dependency is proposed, but these extensions are not yet available in any product. The new design propose a replicable VF that can work in cloned mode to allow read and write of the complete VF state.

4 Comparison of techniques

A comparison of discussed techniques with respect to different requirements is given in Table 1. The requirements not listed in table implies all techniques satisfy those requirements.

The user requirements are met by all proposed techniques. The assignment and management of SRIOV

Requirement	Emulated	Paravirt	SRIOV	Remarks
Equivalence	Yes	Partial	Partial	Equivalence is required at operation level rather than device level
Management	Easy	Easy	Complex	A simplified management interface is a must.
OS dependancy	None	Standard	Standard	The guest OS is assumed to use an abstract device layer
OS integration	Seemless	Module	Module	Loadable kernel module developed by the provider
Resource isolation	Yes	Yes	Yes	It is a must for all methods
Fault isolation	Yes	Yes	Best	Isolation done in hardware for SRIOV NICs
Performance isolation	Yes	Yes	Partial	More VIFs than the maximum VFs supported by SRIOV NICs not addressed
Supportability	Good	Good	Poor	Special hardware for SRIOV and chipset support (IOMMU)
Overhead	High	Medium	Low	Data copy and multiplexing is the main source of overhead
Control Granularity	I/O instr	I/O OP	Assign	This is a price paid for direct assignment of VFs

Table 1: Comparison of requirement adherence for different device virtualization techniques

NICs is little complex compared to other software methods.

Split device I/O model and SRIOV require the OS upper layer stack to use an abstract device independent layer. This is in fact the case in most of the OS network stacks including Linux and Windows. Dynamic module loading depending on the type of device is required for both split device model and SRIOV. This is also supported by all commonly used OS today.

The performance isolation when the number of virtual NICs less than the available VFs is taken care in hardware. But when the number of virtual NICs increase beyond available VFs, some software method is required conjunction with hardware resource division. The QoS features provided by the SRIOV NIC can be used to achieve better fairness. Performance is the best in case of SRIOV NIC. The control granularity of emulated I/O is every hardware state update by the guest VM. For split device model the control granularity is at the interface operation level like send or receive of a packet. SRIOV control granularity is at the VF device assignment level.

5 Open Issues and future work

As we have seen in our previous discussion, no single approach addresses all the requirements for an ideal network device virtualization. We would like to propose an alternate holistic device virtualization architecture that would try to address all the requirements.

5.1 Dynamic capability based network virtualization

Our new architecture proposes a split device model irrespective of available hardware features.A dynamic capability exchange mechanism between the frontend and backend device drivers is proposed to have better resource control. The backend device comes up with a list of capabilities based on the underlying device features, hardware caching features, fairness/QoS settings and device coalesce settings. The capabilities map into different parameters and behavior for different operations in both backend and frontend drivers. The capabilities can change at any point of time because of underlying resource change or change in number of virtual NICs sharing the underlying resource.

The current SRIOV implementation uses device emulation for providing PCI device interface to the guest VM. PCI emulation and interrupt routing overheads could be more than the paravirtualized implementation of SRIOV network device. We plan to implement the paravirtualized model as proposed above and compare the overheads.

Fair resource allocation with SRIOV like devices becomes an issue if the required number of virtual interfaces exceed the available VFs. In a Xen domain-0 device model, the physical function device resource will become

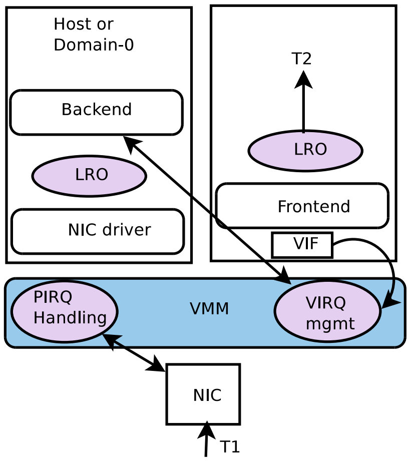


Figure 12: Coalescing at different points of receive path

heavily loaded and the VIFs using older PV driver model would suffer. Dynamic capability model proposed above will make the resource allocation fair and according to the administrator settings using SRIOV rate limiting feature at the run time.

5.2 Other optimizations

We propose *Multilevel adaptive transmit and receive coalescing* for different underlying NIC hardware. Current solutions propose coalescing as an optimization for send and receive aggregation and evaluated them as standalone optimization. Aggregation can be done at different levels (Figure 12) in a virtual NIC scenario. The different points where coalescing is done in receive side are physical NIC coalescing for reducing interrupts, VIRQ coalescing at the driver domain to reduce the number of virtual interrupts, LRO in software to optimize stack receive in both driver domain and guest domain and VIRQ aggregation settings in the guest domain. Given a particular NIC we would like to come up with proper coalesce setting to optimize performance and keep packet processing latency minimal.

Many of the proposed optimization were to improve the cache hit ratio in virtual device operations. There has been a lot of change in both hardware support and cache improvements along with change in hypervisor design. We would like to evaluate the current implementation and all proposed optimization against new hardware features to come up with a list of useful optimization in most commonly used hardware today. Also also want to see the cache behavior change as a function domain-0 physical CPU assignment. We aim to find out the suitable optimizations to take advantage of recent cache improvements.

6 Conclusion

In this report we presented how I/O device virtualization poses different set of challenge than processor and memory virtualization. We classified the requirements based on user expectations at different levels of abstraction. The present techniques like device emulation, para-virtualized split device model and hardware assisted SRIOV like solutions were discussed and evaluated against the requirements. In spite of several research targeted to improve individual problems, all the requirements are not satisfied by any one method. We highlight some areas of research for new design and experimental evaluation that will improve I/O virtualization.

References

[1] Gerald J. Popek, *Formal Requirements for Virtualizable Third Generation Architectures.* Comunications of ACM Volume 17 Issue 7,1974.

[2] P. Barham,B. Dragovic, K. Fraser, S. Hand,T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. *Xen and the art of virtualization* SOSP, 2003.

[3] Keith Adams,Ole Agesen *A Comparison of Software and Hardware Techniques for x86 Virtualization* ASPLOS, 2006.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. *kvm: the Linux Virtual Machine Monitor* Linux Symposium, 2007.

[5] J. Sugerman, G. Venkitachalam, and B. Lim. *Virtualizing I/O Devices on VMware Workstations Hosted Virtual Machine Monitor.* USENIX ATC, 2001.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. *Safe hardware access with the Xen virtual machine monitor.* OASIS, 2004.

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. *REMUS: high availability via asynchronous virtual machine replication.* NSDI, 2008.

[8] Aravind Menon,Jose Renato Santos,Yoshio Turner,G. (John) Janakiraman,Willy Zwaenepoel. *Diagnosing performance overheads in the xen virtual machine environment.* VEE, 2005.

[9] Aravind Menon , Alan L. Cox , Willy Zwaenepoel. *Optimizing network virtualization in Xen.* USENIX ATC,2006.

[10] Yaozu Dong , Dongxiao Xu , Yang Zhang , Guangdeng Liao. *Optimizing Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive Side Scaling.* Proceedings of the 2011 IEEE International Conference on Cluster Computing, 2011

[11] Abel Gordon , Nadav Amit , Nadav Har'El , Muli Ben-Yehuda , Alex Landau , Assaf Schuster , Dan Tsafrir. *ELI: bare-metal performance for I/O virtualization.* ASPLOS, 2012.

[12] Jose Renato Santos , Yoshio Turner , G. Janakiraman , Ian Pratt. *Bridging the gap between software and hardware techniques for I/O virtualization.* USENIX ATC, 2008.

[13] Kaushik Kumar Ram , Jose Renato Santos , Yoshio Turner , Alan L. Cox , Scott Rixner. *Achieving 10 Gb/s using safe and transparent network interface virtualization.* VEE 2009.

[14] Intel documentation, *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology.* http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html

[15] Dong Y., Yang X., Li X., Li J., Tian K., Guan H. *High performance network virtualization with SR-IOV.* HPCA 2010.

[16] LIU J. *Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support.* IPDPS, 2010.

[17] Zhenhao Pan,Yaozu Dong,Yu Chen,Lei Zhang,Zhijiao Zhang. *CompSC: live migration with pass-through devices.* VEE 2012.

[18] Yaozu Dong,Yu Chen,Zhenhao Pan,J. Dai,Y Zhang. *ReNIC: Architectural extension to SR-IOV I/O virtualization for efficient replication.* TACO, 2012

[19] M. Ben-Yehuda, J. Xenidis, M. Mostrows, K. Rister, A. Bruemmer, and L. Van Doorn. *The price of safety: Evaluating IOMMU performance.* OLS 2007.

[20] Carl A. Waldspurger. *Memory resource management in VMware ESX server*, ACM SIGOPS Operating Systems Review, 2002.