A Seminar Report

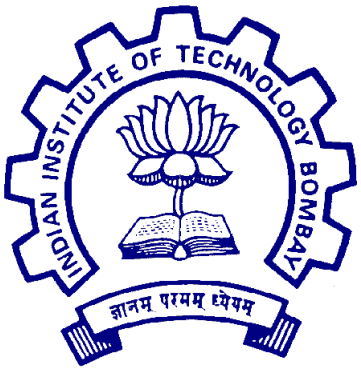
on

Network I/O Virtualization : Challenges and Solution Approaches

Submitted By:
Rinku Shah
Roll No. :134053001

Guided By:
Prof. Umesh Bellur

April 2014



Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

Abstract

Virtualization plays a key role in next generation data centers. In a virtualized system, multiple virtual machines (VMs) share a single physical machine's (PM) resources. Shared resources include CPU, Memory, Disk, NIC, and many more. Virtualization provides benefits like reduced carbon footprint, reduced hardware vendor lock-in, faster server provisioning, and pay-per use model. Virtualized data centers benefit both, the service-provider, as well as the service-user. Service user does not have to invest for the physical infrastructure, also does not have to maintain it. Service provider uses services like VM provisioning, VM consolidation, Memory deduplication, and Disk deduplication to minimize the cost of resources. This eventually also benefits the environment due to reduced carbon footprint. To achieve the benefits of virtualization, improvements on services like VM provisioning, VM consolidation, Live VM migration, and Virtualized I/O access are proposed by researchers.

Virtualization solutions need to ensure *Isolation* and *Efficiency* for the VMs residing on the same PM. To achieve these properties; an abstraction layer called the *hypervisor*, or *VMM (Virtual machine monitor)* is required. This VMM either acts as a layer between the OS and the VMs (Hosted VMM environment), or it is the part of the OS (Bare-metal environment). Abstraction increases overheads for accessing resources. Out of all the resources, I/O resources are the most difficult to partition and manage because:

1. The rate at which events arrive at these I/O devices is very high, as I/O devices are shared by many VMs.
2. I/O devices are comparatively slower than CPUs. According to Moore's law [46], the number of transistors on integrated circuits double every two years, which eventually improves the speed the CPU's. Moore's law was invalidated in 2010, but the fact that the number of cores increase every two years is true. Significant improvement is not achieved by I/O interfaces like PCI, and PCIe. This results in mismatch between CPU and I/O speeds.

Due to above reasons, I/O access becomes a bottleneck.

Network I/O is even more difficult to virtualize as:

1. Packet rates are very high.
2. Receive packet arrival rate is non-deterministic, as packets generally arrive from external hosts.
3. Packets destined for the same VM do not arrive in batches; i.e ; we generally receive an I/O stream with consecutive packets having destination addresses belonging to different VMs.
4. In virtualized scenario, each network packet has to go through an additional layer, VMM. This introduces additional performance overhead.

Due to these reasons, virtualized network I/O access is a challenging problem to solve.

In this report, we perform a detailed survey on *Optimization of Network IO Path in Virtualized Systems*. We survey various sources of network processing overheads and approaches to deal with them. Authors of [39] have done a similar survey on virtualized network I/O access techniques. We have classified I/O optimization techniques with a different view point. High level classification is based on Virtualized I/O access model used; i.e.; whether we use Paravirtualized, Emulated or Direct I/O model. Also [39] has missed out on the approaches like, ELI[14], ELVIS[15], ReNIC[33], vTurbo[28] and many more, that have been proposed after its publication. We also provide a *Comparative Evaluation* of Network I/O Optimization Approaches, which is absent in [39]. We also discuss the *Open problems* in this domain, and come up with some *solution directions*.

Contents

List of Figures

List of Tables

1 Introduction

The notion of virtualization is not new. It already existed in our multi-programming / multi-tasking operating systems and is called *process virtualization*. Its goal is to abstract hardware resources and allow multiple isolated machines to share the system resources. Resources include CPU, memory, disk, NIC and many more. Resouce sharing/isolation is achieved by running processes under the control of software-layer i.e. OS. Now we have a mature picture, where, multiple virtual machines (VMs) concurrently execute on a single physical machine (PM). Now each VM acts like a process. This is called *system virtualization*. This is achieved by running VMs under the control of a software-layer i.e. VMM.

Popek and Goldberg [40] specifies three requirements that a VMM should satisfy. Equivalence/Isolation, Efficiency and Resource control. Equivalence/Isolation - Any VM program that runs under the VMM should have the same effect as if it is running on the PM directly, except for the differences caused due to resource unavailability. Efficiency - VM programs should execute with minimum decrease in speed. Resource control - VMM should have complete control of the system resources.

To improve efficiency, VMM must allow guests to access resources directly. But it is also essential to ensure isolation and correctness in the guest behavior. So VMM allows guest to execute all non-privileged instructions natively in the hardware without its intervention. For privileged instructions guests trap into VMM for aforesaid reasons. CPU intensive workloads (generally do not use non-privileged instructions) can achieve high performance even when executed in a VM. But IO intensive workloads when executed on VMs, suffer performance losses. For example, in Xen [4] shows that guest transmit performance is 20% of native, and receive performance is 33% of native.

It is quite logical to ask a question, Can we use the methods that are used for Process Virtualization to solve the problem of System Virtualization?. It is comparatively easy to virtualize CPU and memory with certain modifications to process virtualization techniques. But virtualization of I/O devices is a challenge because:

1. The rate at which events arrive at these I/O devices is very high, as I/O devices are shared by many VMs.
2. I/O devices are comparatively slower than CPUs. According to Moore's law [46], the number of transistors on integrated circuits double every two years, which eventually improves the speed the CPU's. Moore's law was invalidated in 2010, but the fact that the number of cores increase every two years is true. Significant improvement is not achieved by I/O interfaces like PCI, and PCIe. This results in mismatch between CPU and I/O speeds.
3. IO devices are shared among the guests. Multiplexing/Demultiplexing of guests needs to be safe and consistent.
4. Requires frequent VMM intervention, that eventually leads to, longer IO latency, and more CPU overhead.
5. It is necessary to provide QoS control mechanism, as multiple VMs share same IO device, and some VM may dis-proportionately use IO bandwidth.

NIC device is even more challenging to virtualize because:

1. Most devices like disk or memory use pull approach; i.e. ; Applications make request for a certain disk/memory location. Hence guest OS is aware of the requests which helps in various optimizations like scheduling and batching. This holds true for sending network packets too.

But network packets can be received from outside hosts, so we are unaware about the time of arrival as well as the arrival rate. Hence interrupts are needed where the guest needs to trap back to the VMM.

2. Packet rates are high. Typically packets do not arrive in batches; i.e ; we generally receive an I/O stream with consecutive packets having destination addresses belonging to different VMs.
3. Data must be copied to the guest VM on receive events. An additional data copy is required in case of Xen [1]
4. In virtualized scenario, each network packet has to go through an additional layer, VMM. This introduces additional performance overhead.
5. Certain applications are latency sensitive. These applications need to satisfy SLAs and meet strict deadlines.

There are multiple proposals for *Virtualized I/O Optimization*. This report is organized as follows: Section 2 discusses and compares various I/O virtualization techniques, discusses the network I/O path taken by Xen and KVM hypervisors, and also details out sources of I/O virtualization overhead in Xen and KVM. Section 3 classifies and discusses various *Network I/O Optimization Approaches*. Section 4 discusses the open problems.Section 5 discusses the future work.

1.1 I/O Virtualization techniques

We can classify I/O virtualization techniques as Fully virtualized model, Paravirtualized model, Emulated device model and Hardware assisted model (Direct I/O).

1.1.1 Fully Virtualized device model

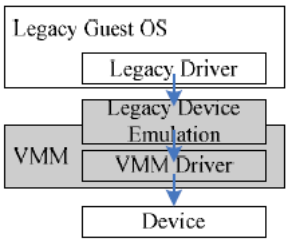


Figure 1: Full Virtualized Device Model (Source: [39])

In the fully virtualized device model [39], VMM is supposed to manage I/O access for the VMs. The guest VM is unmodified and in unaware of the fact that it is virtualized. Whenever I/O operation is requested by the guest application, guest driver initiates a IN/OUT instruction, which is a privileged instruction. Such instructions causes a trap to the VMM. VMM decodes the trapped I/O instruction, maps it to the hardware, and directs the device to complete the execution.

This technique is efficient and transparent. But VMM has to control the devices, and has to do binary translation of I/O requests. If there are changes in the guest driver, VMM driver needs to be modified. This makes the technique complex and difficult to implement. VMware ESX server uses the fully virtualized device model.

1.1.2 Paravirtualized device model

In paravirtualized I/O model [39], device drivers are split into two parts, viz. , Front-end and Back-end driver. Front-end driver is installed in guest VM, whereas Back-end driver is installed in a

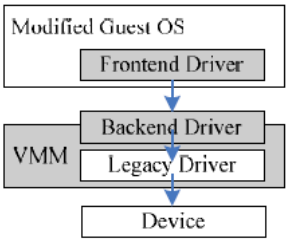


Figure 2: Paravirtualized Device Model (Source: [39])

privileged VM (a.k.a Driver Domain/Domain 0). Guest OS needs to be modified to achieve this. When I/O operation is requested by the guest application, front-end driver forwards the requests to the back-end driver. Back-end driver decodes the requests, maps it to the hardware, and directs the device to complete execution. Back-end driver can help in managing resource-control as well as many optimizations like batching guest requests.
This technique requires guest modification. Several optimizations are implemented to improve its efficiency. Unlike full virtualization, VMM is not involved in device access, hence it is easier to implement paravirtualization. This technique is implemented by Xen.

1.1.3 Emulated device model

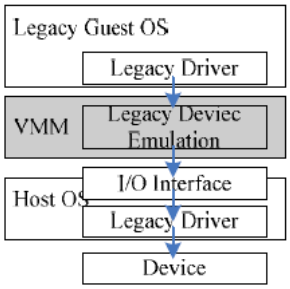


Figure 3: Emulated Device Model (Source: [39])

Emulated device model [39] is used in host based VMM like VMware Workstation and KVM. This VMM is an application that runs on the Host VM. VMM does not have direct access and control of the hardware. VMM relies on the host OS to handle I/O requests. When an I/O operation is requested by guest application, it traps to the VMM, and is passed to the Host OS via a system call. Host OS handles this request.
The efficiency of this technique is lower than the other models, due to context switches. It includes switch between Guest OS and VMM, switch between kernel space (VMM) and user space (emulation application) and switch between user space and Host OS kernel. Several optimizations are suggested to reduce the number of context switches. Guest OS does not require any modification.

1.1.4 Direct device Assignment model

In Direct device assignment model [39] (a.k.a VMM Bypass model), VMM is removed from the I/O access path. This is because of the observation that VMM intervention increases overheads and hampers I/O performance. There are certain challenges to this approach

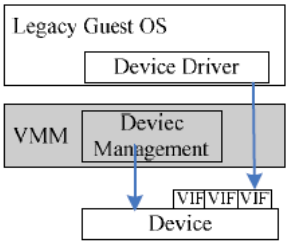


Figure 4: Direct Device Assignment Model (Source: [39])

1. For data copy from device to the guest, DMA operation is required, and for this I/O devices require machine address, which is known only to the VMM.
2. The maximum number of virtual interfaces available is constrained by the hardware.
3. VMM needs to virtualize interrupt for I/O request notifications and I/O completion notifications.

This approach requires to solve two problems, Isolation and Sharing. Isolation means that VMM needs to ensure that VM does not access the memory area that is not allocated to itself. Sharing means that device should provide multiple virtual interfaces for concurrent guest access.To implement this approach, we require hardware assistance like

1. Intel VT-d, AMD IOMMMU - for translation of Guest virtual address to Machine addresses securely.
2. NIC devices with SR-IOV (Single Root IO virtualiztion) or MR-IOV (Multi Root IO virtualization) - for exposing multiple virtual interfaces to the VMs.

This model is highly efficient, as VMM is bypassed from the I/O path of latency-sensitive packets.
The downside of this approach is –

1. It is not scalable.
2. VMM does not have control over traffic generated by VMs.
3. Live-migration is a challenge, as it is difficult to migrate hardware state, at high frequency packet rates.

1.1.5 Comparison

Various I/O virtualization techniques are studied, and their comparative evaluation is shown in Table 1.

1.2 Virtualized Network-device Access Procedure

In order to improve I/O efficiency, it is necessary to analyze the sources of overhead. This report considers two VMMs, Xen and KVM as they are open source.

1.2.1 Xen [1]

Xen presents a paravirtualization as well a full virtualization solution. Xen has a special privileged domain (a.k.a Driver domain/DOM0) that has direct access to the hardware, and is also used to manage the virtual machines. DOM0 runs in a higher CPU privileged mode as compared to other virtual machines.

Table 1: Comparison of I/O Virtualization techniques

	Full Virtualized model	Paravirtualized model	Emulated model	Direct Device Assignment
Approach	Binary Translation and Direct Execution	Hypercalls	Exit to Root mode on privileged instruction execution by guest	Doorbell interrupts and use of private hardware channels
Efficiency	High	High	Low	High
I/O Latency	High	High	Very High	Low
CPU Consumption	Very High	Very High	Very High	Low
Guest modifications?	No	Yes	No	–
Fault Isolation	Good	Good	Good	Best
Scalable?	Yes	Yes	Yes	Limited
Live-Migration supported?	Yes	Yes	Yes	Challenging
Management Complexity	Low	Low	Low	High
Approach implemented by	VMware ESX server	Xen	KVM, VMware Workstation	Xen, KVM, ...

Xen Network I/O communication model

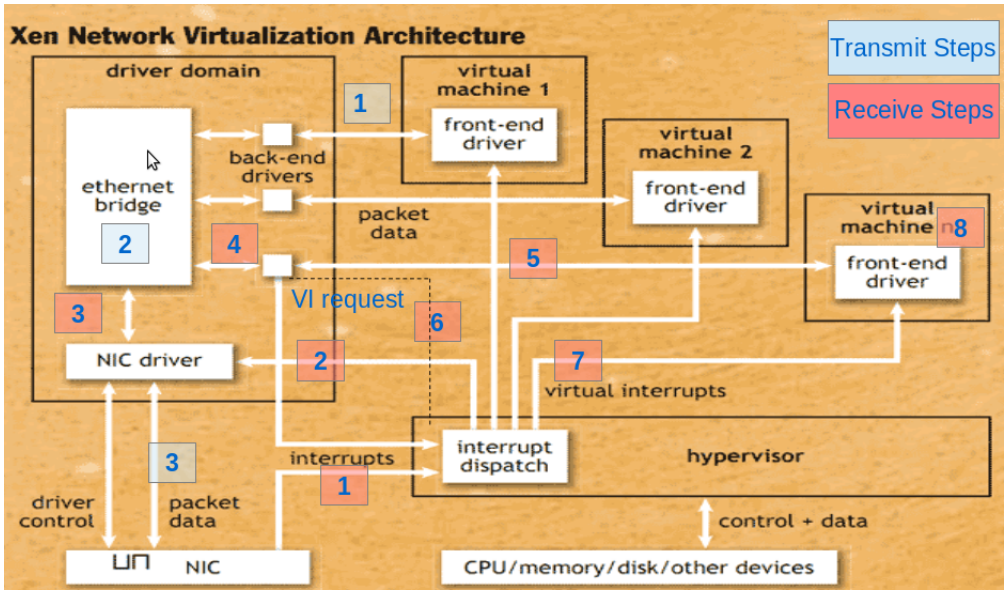


Figure 5: Xen Network I/O Transmit/Receive processing path (Source: Network Virtualization: Breaking the performance barrier, ACM queue, 2008)

A packet transmission takes place according to the following steps:

1. Packet copy/remap from VMs front-end driver to Driver domains back-end driver
2. Route the packet through ethernet bridge to physical NIC's driver
3. Enqueue packet for transmission on network interface

A packet reception takes place according to the following steps:

1. NIC generates interrupt; and is captured by hypervisor
2. Captured interrupt is routed from hypervisor to NIC's device driver in driver domain as virtual interrupt
3. NIC's device driver transfers packet to eth bridge
4. Bridge routes packet to appropriate back-end driver
5. Back-end driver copies/remaps packet to front-end driver in target VM
6. Back-end driver requests the hypervisor to send virtual interrupt to front-end driver in target VM
7. Hypervisor sends virtual interrupt to front-end driver of target VM
8. Front end driver delivers packet to VM's network stack

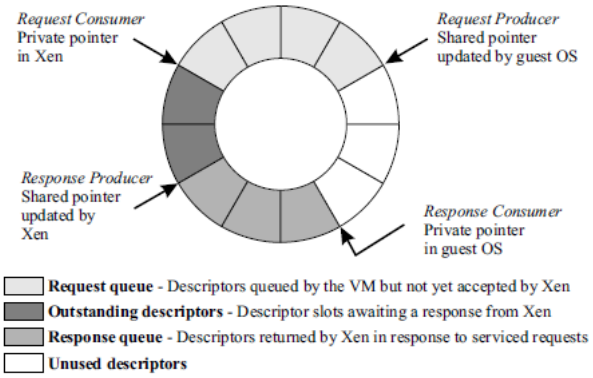


Figure 6: Structure of asynchronous I/O rings; which are used for data transfer between Xen and guest OS (Source: [1])

Netfront [11] uses shared memory I/O channels for communication with the netback driver. Each I/O channel comprises of an event notification mechanism, and bidirectional ring to handle asynchronous requests and responses. Event notification enables netfront and netback to trigger a virtual interrupt; whereas I/O ring carries I/O buffer descriptors. To enable driver domain to access I/O buffers in the guest, page grant mechanism is provided. Guest creates a grant reference and forwards it as a part of I/O buffer descriptor. Driver domain invokes a hypercall using this reference and accesses the guest page. Driver domain uses a hypercall to map guest page into its address space before sending the packet. After NIC sends it, page is made free, and grant is revoked by the device driver. Similarly on packet reception netback invokes a grant copy hypercall to copy packet to the guest page.

Sources of Overhead

Aravind Menon [4] broadly classifies overheads as per-packet and per-byte. Per-packet overheads include tasks like packet header processing, buffer management, and IO channel transfer . Per-byte overheads include tasks like data copy, checksum computation. [4] provides a detailed classification with transmit/receive overheads as shown in Table 1.

Aravind Menon [4] has found through experiments that

1. Guest Transmit performance is 20% of native and Receive performance is 33% of native.

Table 2: Classification of overheads in Xen

Overhead class	Details	TX overhead	RX overhead
per-byte	Data copy routines • TX: copy from guest domains file system buffers to front-end drivers socket buffers • RX: 1. copy from driver domain to guest domain 2. copy from guest kernel to guest application	14%	14%
non-proto (per-packet)	Bridge and Netfilter routines	15%	19%
netback (per-packet)	Initializes 1. Transfer of ACK packets from driver to the guest for transmit workloads 2. Transfer of receive packets from driver to the guest for receive workloads	14% (inclusive of netfront)	18% (inclusive of netfront)
netfront (per-packet)	Initializes 1. Transfer of data packets from guest to the driver for transmit workloads 2. Transfer of ACK packets from guest to the driver for receive workloads	included in netback	included in netback
tcp rx and tx(per-packet)	Transmit and receive TCP routines in the guest	7%	10%
buffer (per-packet)	Buffer management routines and Linux sk_buff structure in both driver and guest domain	12%	12%
driver (per-packet)	Device driver running in driver domain	11%	8%
xen	Domain scheduling, inter-domain interrupts, validation of packet transfer rights, ...		
misc	other routines that cannot be classified as per-packet or per-byte		

2. 62% of total CPU time is spent in guest domain, out of which 35% in network virtualization stack in driver domain and 27% in Xen hypervisor. 38% of time spent in network processing in guest.
3. Number of L2 cache misses are 25 times more than bare-metal.
4. Number of Instruction TLB misses are 2.5 times more than bare metal.
5. Number of Data TLB misses are 3 times more than bare-metal.

In Xen paravirtualized I/O model, following are the reasons of overhead observed by [11].

1. In native Linux, data is copied only once, Kernel socket buffer to application buffer; whereas in Xen PV driver model, data is copied twice, kernel memory of the driver domain to kernel memory of the guest, and then to application buffer
2. Copy overhead occurs because of different memory address alignments for source and destination buffers. Intel processors are efficient if source and destination memory locations have the same 64-bit alignment, and same cache line alignment. But received packet data is non-aligned, as IP header requires alignment for efficient parsing. When copying the packet to the guest, header starts at 64-bit word boundary, but data is mis-aligned. These two mis-aligned data copies requires 4.5 times more CPU cycles than native Linux [11].
3. Xen uses TCP segmentation offloading (TSO), and Large receive offloading(LRO) in its virtual interfaces to process large packets, and improve the packet processing efficiency. To ensure this, netfront posts full page buffers, which are used as fragments in socket buffers. Netfront copies the first 200 bytes in the main linear socket buffer area, rest data remains as the fragment. This increases copy overheads and buffer allocation overheads.
4. Most of kernel overhead is due to bridge and netfilter cost.
5. Hypervisor pins source and destination page to prevent a page being freed when grant is active. The pinning process requires large number of CPU cycles.
6. 17% of the total CPU cycles are wasted for grant operation functions. To revoke grant, atomic compare and swap instructions are needed and are costly. Status bit (GRANT IN-USE/NOT) updated by Xen and GRANT ACCESS PERMISSION bit updated by guest are to be checked atomically.

1.2.2 KVM [2]

KVM (Kernel virtual machine) provides a full virtualization solution for a x86 hardware that has virtualization support like Intel VT-x and AMD-V. KVM uses the emulated device model for I/O device virtualization. KVM has three modes of operation; user mode, kernel mode, and guest mode. KVM consists of a loadable kernel module that provides the virtualization infrastructure. As the VMM (i.e. KVM) is a part of the kernel, it does not have to implement scheduling and memory management functions of a running guest OS. It treats running VM as process, and uses Host Linux functions for scheduling and memory management. This also makes KVM a thin layer. VMM and Host OS are part of kernel mode, hence switching overheads would also decrease. KVM requires a modified QEMU in the user space to handle I/O operations. QEMU is an open source machine emulator, and virtualizer. QEMU can execute programs written for one machine on a different machine using dynamic translation.

KVM-HVM (Hardware virtual machine) Network I/O communication model

KVM uses user-space QEMU application for each VM, to handle I/O requests and responses. TAP device is a software that performs all the tasks of a hardware NIC, i.e., it works like a layer 2 device, but in software. QEMU application binds itself to the TAP device which is connected to the physical NIC through the bridge software provided by Linux.

When a packet is received by the physical NIC, the following events occur

1. A physical interrupt arrives at the host, which is handled by the NIC driver.
2. NIC driver passes the packet to the virtual ethernet bridge. It does demultiplexing on the basis of MAC address and passes the packet to the appropriate TAP device which implements NIC functions in the software.

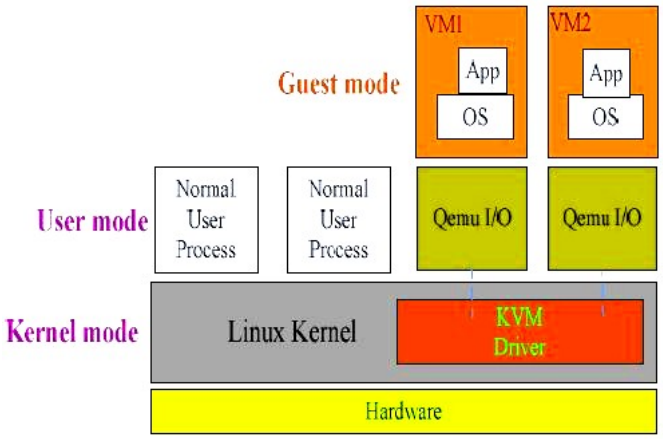


Figure 7: KVM based architecture (Source: [42])

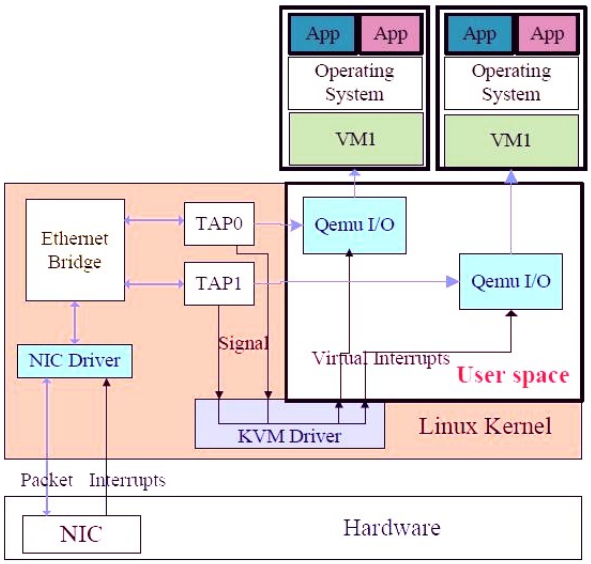


Figure 8: KVM Network I/O architecture (Source: [42])

3. TAP device forwards the received packet to the QEMU, that is bound to this tap device.
4. After packet transfer, tap driver signals the KVM driver to send a virtual interrupt to the correponding guest notifying for a new packet.
5. On receiving the virtual interrupt, QEMU forwards the packet to the guest network stack.

I/O path is very long, requires multiple context switches (VM ENTRY/VM EXIT) , multiple data replication, and QEMU is a pure software. All these overheads results in reduction of KVM I/O performance, i.e., KVM provides the network throughput of about 15% of bare-metal. KVM paravirtualized guest drivers can significantly improve the efficiency (especially I/O efficiency). KVM currently adopts Virtio[3] which is the standard framework for Linux device drivers.

KVM-virtio (Paravirtualized machine-PVM)[3][43]

As mentioned above, KVM-virtio is the paravirtualized I/O model, that is developed to improve virtualized I/O efficiency. Similar to Xen, KVM-virtio uses a split driver model. It splits guest device driver into front-end and back-end driver. Front-end drivers are implemented in the guest OS whereas back-end drivers are implemented in the hypervisor.

Top level layer virtio is a virtual queue interface (a.k.a. virtqueue) that acts as an interface between back-end and front-end drivers. Drivers can use zero or more virtqueues, for eg. network driver needs two queues, one for transmit and other for receive. Virtqueues are implemented as a ring (similar to Xen), for transport of packets from Guest to hypervisor and back.

KVM-virtio Network I/O communication model

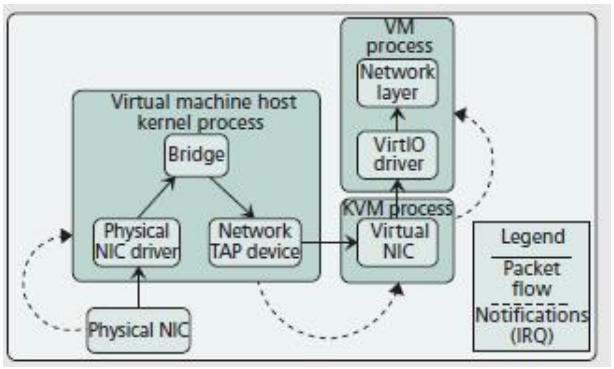


Figure 9: KVM-virtio network I/O path (Source: [41])

When a packet is received by the physical NIC, the following events occur

1. A physical interrupt arrives at the host, which is handled by the NIC driver.
2. NIC driver passes the packet to the virtual ethernet bridge. It does demultiplexing on the basis of MAC address and passes the packet to the appropriate TAP device which implements NIC functions in the software.
3. TAP device forwards the received packet to the VNIC driver, that is bound to this tap device.
4. After packet transfer, tap driver signals the VNIC driver to send a virtual interrupt to the correponding guest notifying for a new packet.
5. On receiving the virtual interrupt, VNIC forwards the packet to the guest network stack.

Sources Of Overhead

Authors of [44] performed a test that consists of a host machine with 2 GB memory, 2 Gb network connection, KVM version 70, QEMU emulator version 0.9.1. On the machine without the virtio driver, the QEMU emulation of Intel E1000 driver (1 Gb network connection) was used. The OS on the host and the guest machine is CentOS 5 with Linux kernel version 2.6.26.2. This Linux version includes the virtio drivers for the paravirtualized guest machine.
The TAP latency on sending path is composed of the following functions:

1. tun_get_user: It gets a network packet from the user space buffer (QEMU).
2. tun_net_xmit: It transmits the packet to the bridge.

The TAP latency on the receiving path is composed of the following functions:

1. tun_get_user: It gets a packet from the user space buffer (by the bridge-utils).
2. tun_net_xmit: It transmits the packet to the QEMU process.

Timeline of a network packet transmission [44] from guest machine with virtio and emulated network drivers is shown below.

KVM-HVM model	
Steps involved	Time for step in μsec
VM EXIT	0.58
KVM Interrupt Handling	0.3
QEMU Emulation	3.02
TAP Transaction	3.47
Bridge Transaction	1.72
VCPU load	0.21

Table 3: Classification of transmit overheads in KVM-HVM model

KVM-virtio model	
Steps involved	Time for step in μsec
virtio_add_buff	0.26
virtio-net:handle tx	1.71
TAP Transaction	3.32
Bridge Transaction	1.72

Table 4: Classification of transmit overheads in KVM-virtio model

The KVM-HVM model takes 3.02 μs while the KVM-virtio takes only 1.97 μs; i.e.; an improvement of 35%. Timeline of a network packet reception [44] from guest machine with virtio and emulated network drivers is shown below.

KVM-HVM model	
Steps involved	Time for step in μsec
Bridge Transaction	1.72
TAP Transaction	2.7
QEMU Emulation	2.57
KVM interrupt handling	0.3
VCPU load	0.21

Table 5: Classification of receive overheads in KVM-HVM model

KVM-virtio model	
Steps involved	Time for step in μsec
Bridge Transaction	1.72
TAP Transaction	2.56
virtio backend	0.78
virtio frontend	0.35

Table 6: Classification of receive overheads in KVM-virtio model

The KVM-HVM model takes 2.57 μs while the KVM-virtio model takes only 1.13 μs; i.e.; an improvement of 56

The total packet receiving latency in KVM-virtio model is about 5.47 μs; this time is 24% lower than the time required to receive a packet on KVM-virtio guest. The paravirtualization time of a packet (virtio frontend time + virtio backend time) is 1.13 μs which is less than half of the 2.57 μs of the emulation time of KVM-HVM model.

The major source of overhead in KVM-HVM model is QEMU emulation time. The major overhead in general for both HVM and PVM approach is TAP transaction, that involves the hypervisor in the I/O path.

An experiment was carried out by [44], for computing Network bandwidth and the following results were obtained.

KVM Network Bandwidth		
	KVM-virtio driver model (in Mb/s)	KVM-HVM model (in Mb/s)
Transmit bandwidth	286	181
Receive bandwidth	570	371
Transmit CPU utilization	98.1	96.3
Receive CPU utilization	97.3	95.7

Table 7: KVM Network Bandwidth

The improvement of KVM-virtio driver over KVM-HVM model on the transmit bandwidth is 54% and on the receive bandwidth is 58% [4]. CPU utilization was found to be little higher for KVM-virtio as compared to KVM-HVM

Performance of various hypervisors

Experimental results comparing the performance of various hypervisors was reported in [9]. Figure 10 shows results for Netperf, a simple network benchmark that uses a stream of TCP packets to evaluate the performance of data exchange. This comparison would help us in choosing an appropriate hypervisor that satisfies application requirements.

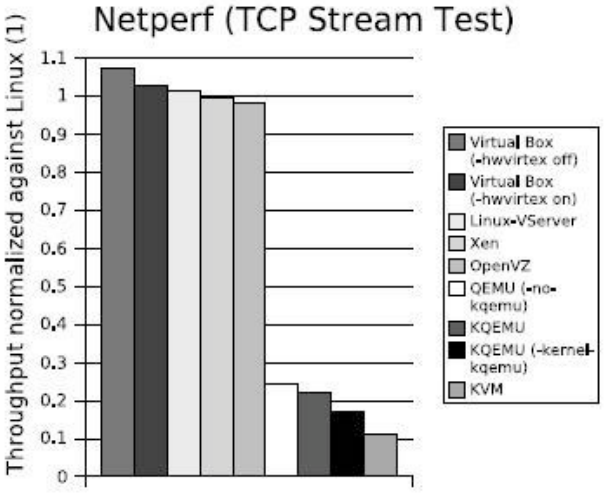


Figure 10: Comparison of Network Performance provided by various Hypervisors (Source: [9])

- It can be observed that, technologies based on QEMU provide poor network I/O performance compared to others.
- The performance of VirtualBox is even better than Linux, possibly due to the use of a special network driver implementation that communicates closely with the physical network interface.
- With PV, Xen could attain near native performance
- Native KVM, with trap-and-emulate, shows performance of 11%

2 Network I/O optimization Approaches

We can broadly classify Network I/O path optimization techniques into Software and Hardware-assisted approaches. They are further classified on the basis of the virtualization technique used by VMM, i.e., Paravirtualized model, Emulated model and VMM-bypass Direct I/O model.

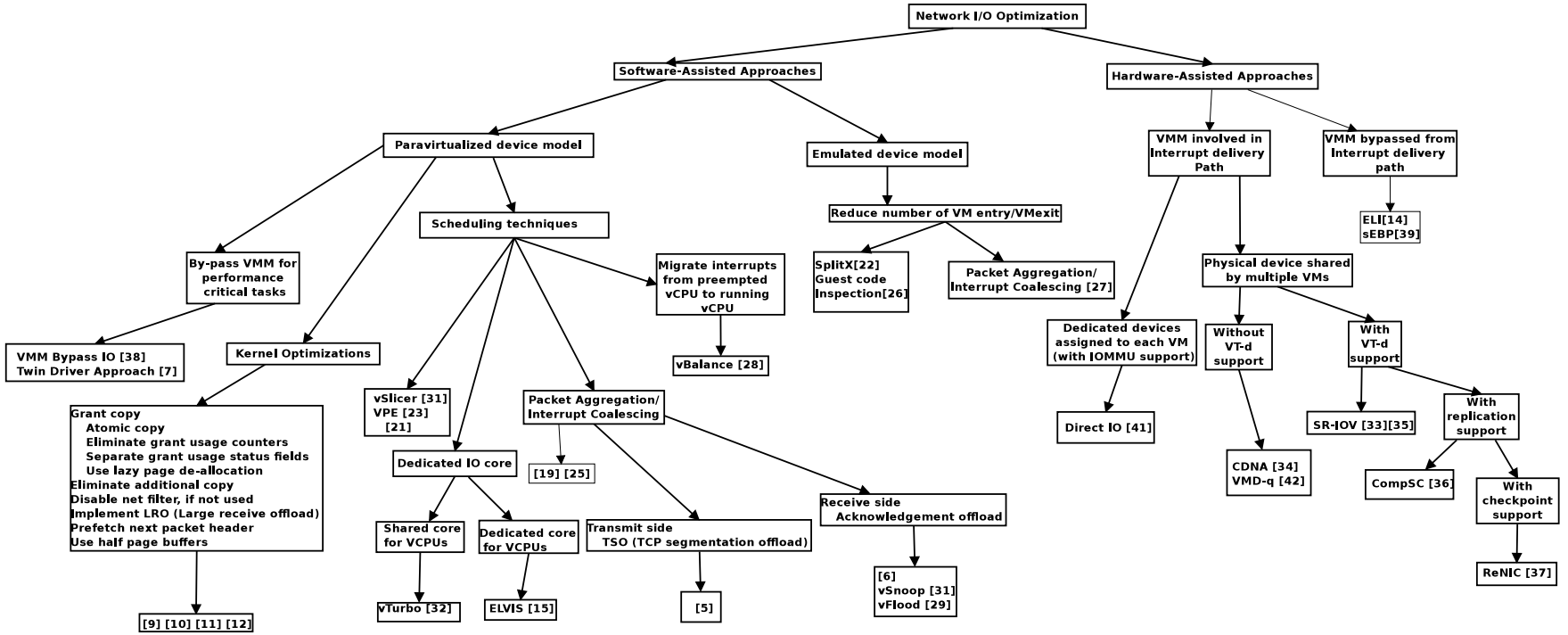


Figure 11: Taxanomy of Network I/O Optimization Approaches

2.1 Optimizations for Paravirtualized device model

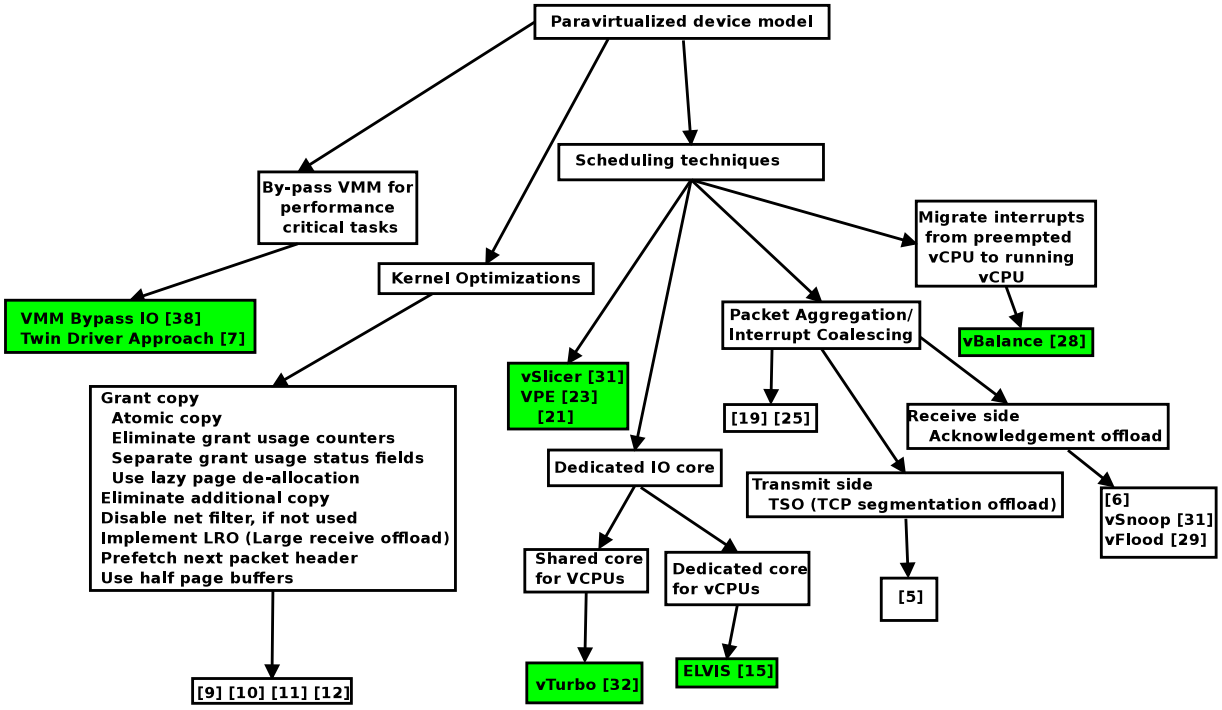


Figure 12: Classification of Network I/O Optimization Approaches for VMM using Paravirtualized device model

2.1.1 Eliminate VMM from the I/O path of performance critical tasks

One of the major reasons of performance degradation is that the VMM increases length of I/O path. One of the optimization approaches proposed (VMM Bypass[34]) is to eliminate VMM from the I/O path.

VMM-bypass [34] approach is inspired by OS-bypass design. In OS-bypass, user processes are allowed to access I/O devices directly and safely, without going through the operating system. Devices allow OS bypass for frequent and time-critical operations, whereas configuration and management operations have to go through the kernel. To ensure safety, OS-bypass capable device provides virtual access points to user applications. These virtual access points are encapsulated in separate I/O pages, and mapped to virtual address spaces of different user processes. VMM bypass uses the idea of paravirtualization to implement technique similar to OS-bypass.

1. Guest VM implements a device driver (a.k.a. guest module) to virtualize the I/O device and handle privileged I/O access.
2. This driver creates virtual access points and maps them into virtual addresses of user applications.
3. The guest module cannot directly access the device, so a backend module is created . This module can provide hardware access to different guest modules either through the VMM or directly.
4. Backend module acts as a proxy as well as an co-ordinator among different VMs.

Limitations of this approach are:

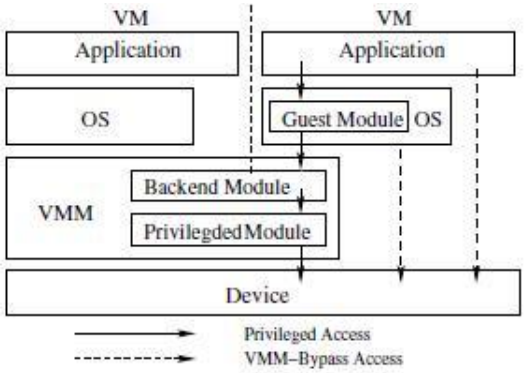


Figure 13: VMM Bypass I/O VMM directly handles I/O (Source: [34])

1. A Para-virtualized device does not conform to existing hardware interface, this leads to compatibility issues.
2. Placing the driver software in the VMM, increases the code base of the VMM, maintainence becomes difficult.
3. This could corrupt the VMM, if the driver software is corrupt.
4. With new devices, there is a need to introduce new device drivers for the VMM, and also distribute the new VMM version

Twin Driver approach [7] also aims at elimination of VMM from the latency-sensitive I/O path. [7] improves performance, maintain safety and provide software engineering benefits. Authors of [7] propose two drivers, first is the original driver (a.k.a. VM instance), and second is semiautomatically produced from first driver by binary rewriting. (a.ka. Hypervisor instance) Second driver is responsible for safety and efficiency. Both drivers execute concurrently with single instance of driver data. VM instance takes care of performance critical operations of device driver, whereas Hypervisor instance handles operations such as device configuration, management, and error handling. Hypervisor instance can access data while running in the guest context using address translation scheme called Software Virtual Memory (SVM).

SVM allows hypervisor instance to access Dom0 data structures from any guest domain address space. SVM employs runtime address translation and protection mechanism that is incorporated in the hypervisor driver during binary rewriting of VM driver. SVM is a *software translation table (stlb)*, which maps from virtual memory page address in Dom0 address space to mapped virtual page address in hypervisor address space. SVM eliminates context switch. Throughput of this approach scales by 2.4x for transmit and 2.1x for receive workloads.
The limitation of this approach is that some amount of additional software engineering effort is required for binary rewriting. This rewriting needs to be done, everytime there is a driver update.

2.1.2 Kernel optimizations to reduce packet transmit/receive delays

Authors of [9][10][11][12] propose the following kernel optimizations to improve I/O performance and reduce CPU utilization.

1. Additional data copy can be avoided with hardware support. We can make use of multiqueue devices where NIC can place the received packet directly into guest buffer.
2. Copy data between 64-bit aligned memory locations. Overhead further reduces if source and destimation buffers have same cache-line alignment. Both together can reduce the CPU cost by

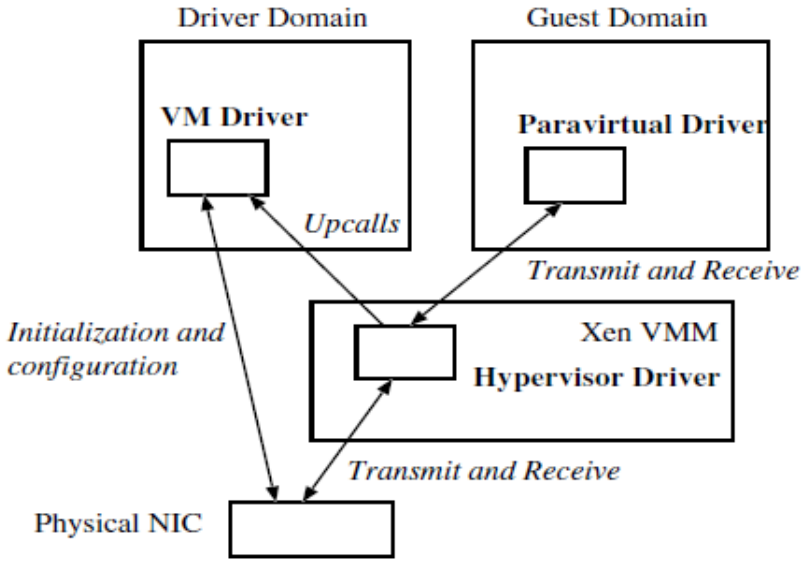


Figure 14: Twin-driver architecture (Source: [7]

a factor of two as compared to Direct I/O.

3. Most of the bridge and netfilter cost can be eliminated by appropriately configuring the kernel when netfilter rules are not used.
4. Pinning cost can be reduced by using grants for data copy.
5. To ensure atomicity for grant revoke and make it less expensive, store two grant bits in different words and use memory barriers.
6. To reduce CPU cycles on acquiring and releasing spin locks, combine multiple grants in a single critical section.
7. Move grant copy operation from driver domain to guest domain. This improves cache locality for second data copy.
8. Reuse page grants. Keep the guest I/O buffers mapped in driver domain even after I/O is completed.

The results of various optimizations are plotted by [11] as shown in figure 15.

2.1.3 Scheduling techniques

Some optimization approaches explore various hypervisor scheduling techniques for VMs to improve I/O performance. Some techniques aim at improving throughput; while others aim at reducing IRQ processing delay (indirectly reducing scheduling dealys) for latency sensitive VMs (LSVM).

2.1.3.1 Reduce interrupt processing delays Hypervisor scheduling could increase scheduling latency of vCPU; which in turn increases processing delays. In case of SMP machines, vBalance [24] uses limited help from hypervisor and modifies the guest to migrate interrupts from a pre-empted vCPU to the running vCPU.
An SMP aware OS has the capability of executing a process/thread on a specific CPU. It makes use of IO-APIC. IO-APIC contains a redirection table that routes interrupts to the relevant cores, by

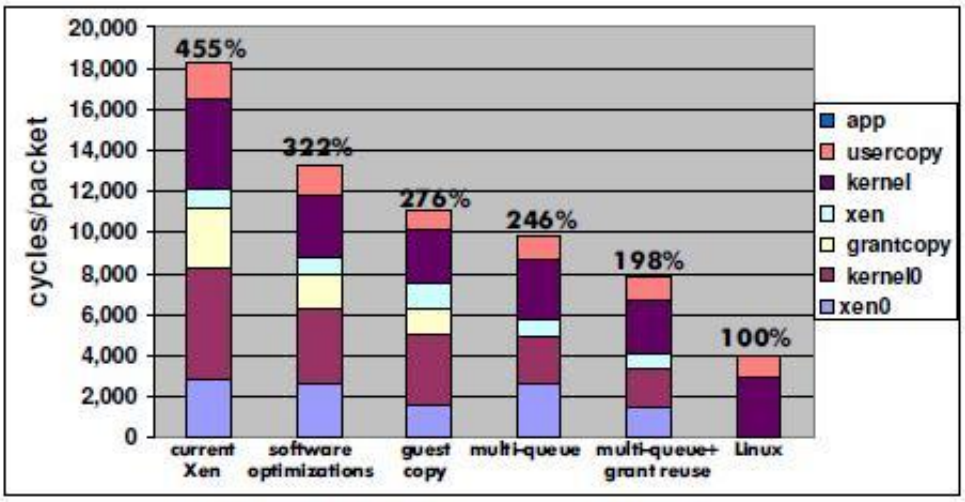
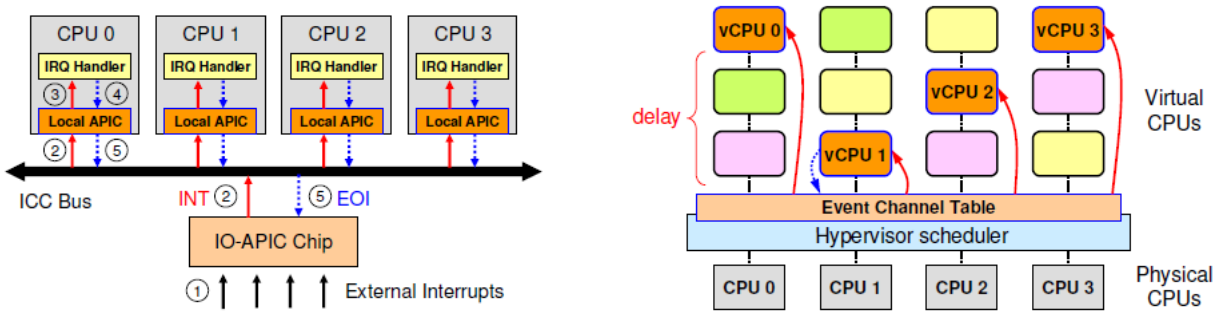


Figure 15: Xen PV driver Optimizations (Source: [11])

writing an interrupt vector to LAPIC. IO-APIC can be used to balance interrupt load using interrupt balancing software like irqbalance in Linux. In paravirtualized-Xen the function of IO-APIC chip is



(a) IO-APIC for SMP physical machine

(b) Event channels for SMP virtual machine

Figure 16: Comparison of I/O machanisms between physical SMP and virtual SMP (Source: [24])

virtualized using event channel. For each VM, global event table is maintained, which consists of details like event type, event state, and notified vCPU. Guest OS informs the hypervisor about which event-type has to be bound to which vCPU.

In case of a physical SMP, when an event is received by LAPIC, it jumps to the interrupt handler instantly. In a virtual SMP, vCPU to which an interrupt is mapped may not be running and time of event handling depends on hypervisor scheduling, leading to delays. In physical SMP, interrupts can be delivered to a set of cores; whereas in virtual SMP event channel method, interrupts can be delivered to a single vCPU. In physical SMP, idle process consumes unused CPU cycles; whereas in virtual SMP, unused CPU cycles are taken away by the hypervisor and resources assigned to current VM get wasted. Guest module receives scheduling information about the vCPUs from the hypervisor through shared memory. IRQ load monitor analyzes current interrupt load statistics for each vCPU, and uses it to decide if there is an interrupt imbalance. Once interrupt imbalance is identified, Balance Analyzer identifies the vCPU to schedule the interrupt on using sched_info, and IRQ Map Manager does the remap.

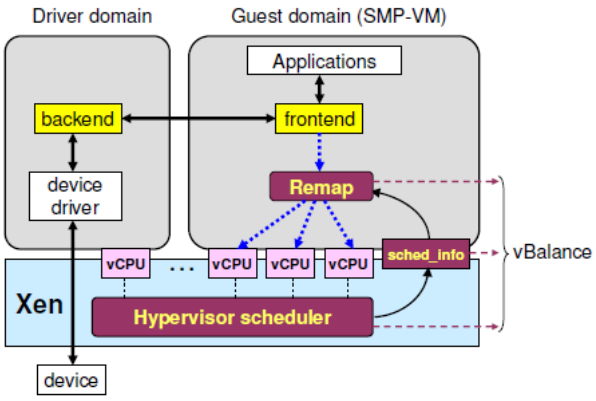


Figure 17: vBalance architecture (Source: [24])

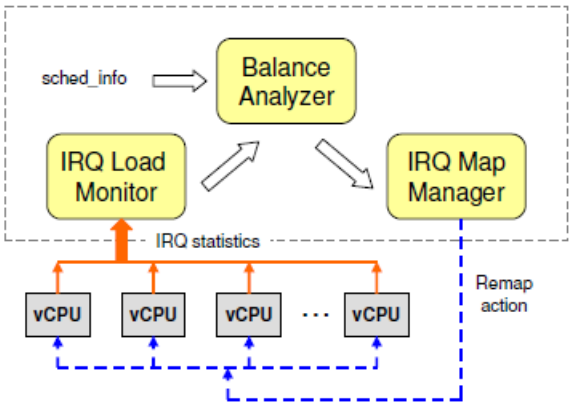


Figure 18: vBalance remap module in guest OS (Source: [24])

vBalance reduces network latency, increases throughput, and balances interrupts on vCPUs well. But it requires communication of hypervisor scheduling information to the guest. This might lead to some security threat.

2.1.3.2 Reduction in response time vSlicer [27] proposes hypervisor level scheduling that schedules LSVMs more frequently but with smaller micro time slices. Polling LSVMs more frequently leads to reduction in interrupt delivery time and processing time; which in-turn reduces response time. Assuming VM time slice as 30 ms, worst case scheduling latency for I/O request is 90 ms and response time is 100 ms. Using vSlicer, it is possible to reduce response time to 40 ms. It is necessary to decide on time slice value for LSVM; if it is too small, number of context switches will increase; hence vSlicer decides time slice as 5 ms. It is also required to decide the schedule. For this purpose, number of LSVMs and non-LSVMs (NLSVM) are considered, and also number of times we wish to run a particular VM during one slice is considered. For example, for two LSVMs and two NLSVMs, with 3 scheduling turns during one slice can be seen in figure 19 (b).
The limitation of this approach is to determine the VMs that can be grouped as LSVMs, and configure the scheduler accordingly. It reduces scheduling latency, but does not completely eliminate it. It also does not consider workload frequency.

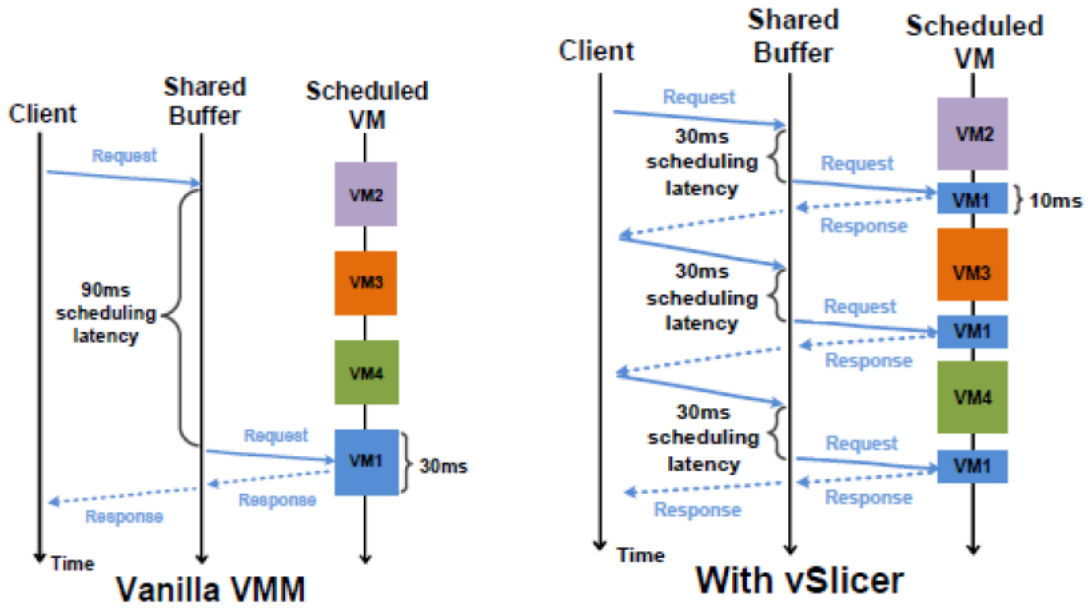


Figure 19: Response time for a) Vanilla VMM b) vSlicer (Source: [27])

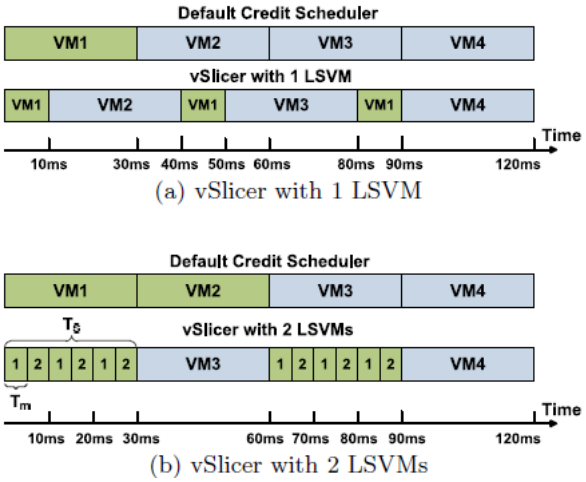


Figure 20: vSlicer Scheduling sequence (Source: [27])

2.1.3.3 Dedicated I/O scheduling cores In SMP machines, each core maintains its *Interrupt Descriptor table (IDT)*. In a typical x86 system, physical interrupt arrives at the hypervisor, handled by Host IDT. If host determines that the interrupt belongs to a guest, it injects a virtual interrupt to the guest causing VM entry (context switch), and the interrupt is handled by guest IDT. If guest wishes to send I/O request notifications to the hypervisor, VM exit (context switch) is caused.

In traditional paravirtual systems, there are two exits caused for I/O operations, one for I/O request notification and other for I/O completion notification. ELVIS [15] aims at eliminating both the exits on a paravirtual system, so that the I/O performance improves. ELVIS executes hosts and guests on separate and dedicated cores to avoid exits caused by I/O operations. It proposes a fine

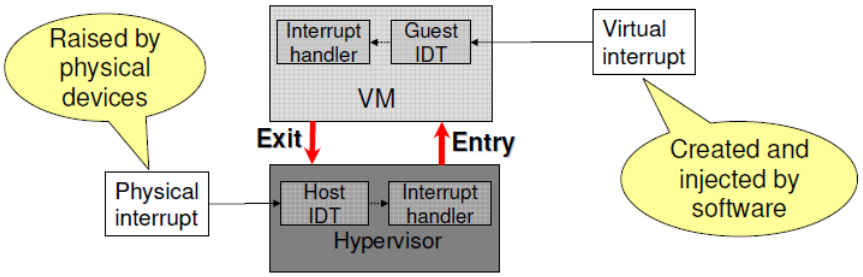


Figure 21: x86 interrupt handling (Source: vTurbo presentation)

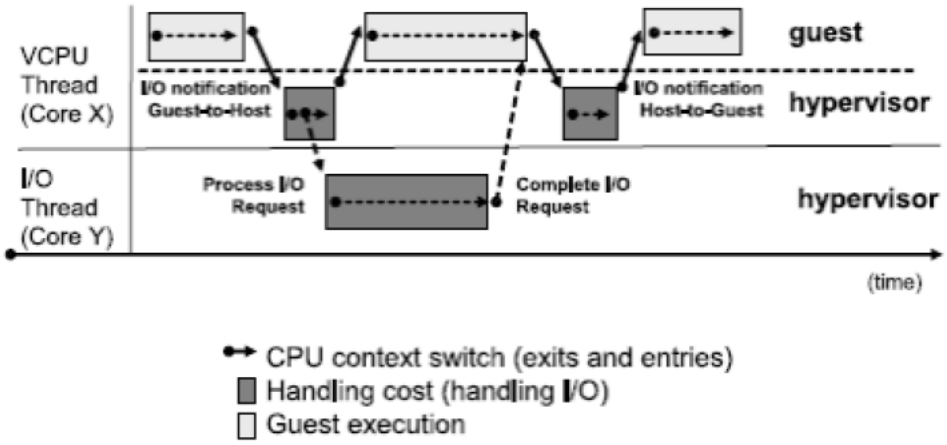


Figure 22: Traditional Paravirtual I/O (Source: [16])

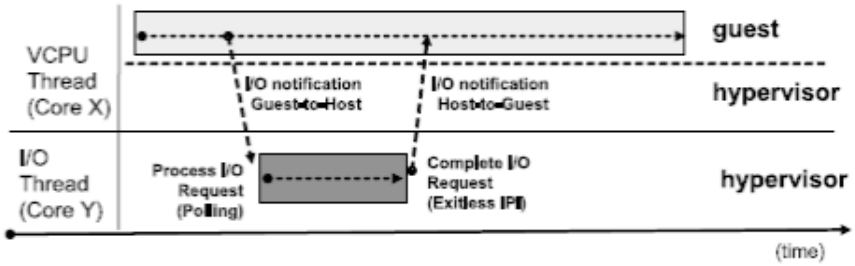


Figure 23: ELVIS communication (Source: [16])

grained I/O scheduler where every guests I/O thread runs on a separate I/O core (Traditionally each guest has its own I/O thread running on different cores). This thread can manage I/O requests of multiple VMs without exits. Further advantage is we can provide fair sharing of I/O thread among various guests as the control is central. As we avoid guest exits, context switches reduce and cache hits also increase.

To ensure Exitless Request Notifications, guest writes request in host‘s shared memory, separate I/O core polls this shared memory. Hence exits are avoided by polling host’s I/O core. To reduce unsuccessful polls,switching between polling and exit based traditional system can be done. To ensure Exitless Reply/Completion Notifications, ELVIS allows I/O core to inject interrupts into another core (eg. Posted interrupts in x86) without causing an exit. ELVIS emulates *posted interrupts* by using

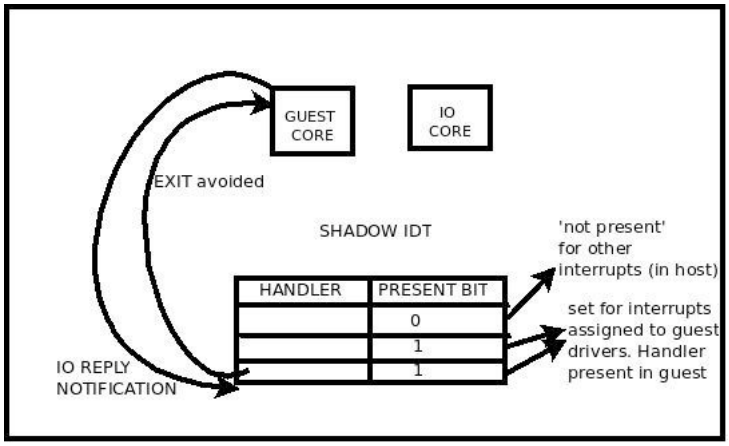


Figure 24: ELVIS emulation to demonstrate Exitless reply notifications

the shadow table technique of ELI [14], because the hardware solution of posted interrupts was in production, and unavailable to the authors. The Shadow IDT technique also aims at eliminating exits, that occur due to request/reply notifications. Shadow IDT is the copy of guest IDT with an additional bit to indicate presence/absence of guest interrupt handling routine. If the entry refers to an interrupt assigned to guest driver, present bit is set; else it refers to sensitive tasks, present bit is set to 0, causing Not Present(NP) exception, further leading to an exit to the hypervisor.
Single CPU core is assigned for hypervisor, and single I/O thread services requests of all VMs. The idea is that on a socket having 8 core CPU, 7 cores are dedicated to 7 VMs and 1 core dedicated for I/O core, so that cache locality would improve performance. ELVIS proposes one I/O core for 7 VMs. If number of VMs increase beyond 7, it proposes a second I/O core for next 7 VMs. The idea is to have the I/O thread running on the same CPU socket where the guest is running, to improve cache hits.
ELVIS does not allow vCPUs to share same core, as sharing could lead to multiple context switches during I/O completion notification, and increased performance overhead. ELVIS does not exactly specify when to switch from polling to exit based traditional scheme. With polling; application receives best-effort service. Size of virtqueues requires some dynamic setting, as they might overflow if I/O frequency is high.

For those scheduling techniques where the physical core is shared by many vCPUs, each vCPU is assigned fair time-slice (generally 30ms for Credit scheduler). It can be observed that UDP receive suffers from packet losses due to shared buffer full condition; whereas TCP scheduling delays increase due to IRQ processing delays.
vTurbo [28] presents a solution where physical core can be shared by vCPUs. As in ELVIS, vTurbo offloads I/O processing to a dedicated core with scheduling time slice of 0.1 ms. It states that I/O processing involves two basic stages; synchronous IRQ processing, and data copy from kernel to user buffer. vTurbo focuses on reducing IRQ delays.
It proposes a high-frequency scheduling CPU core called turbo core, and a turbo vCPU. Turbo vCPUs of each VM are pinned to the turbo core. It proposes a scheduling policy where time-slices of normal cores (30ms) are different from that of turbo cores(0.1ms), in order to reduce scheduling latency. With the turbo core feature, UDP receive speeds up without packet losses as IRQ latency decreases; whereas TCP processing delay is still high. When user process calls recv(), socket structure is locked to prevent IRQ threads to modify it. TCP acknowledgements can be sent only when application receives data. Solution to this problem is to enable ACK generation from iRQ context

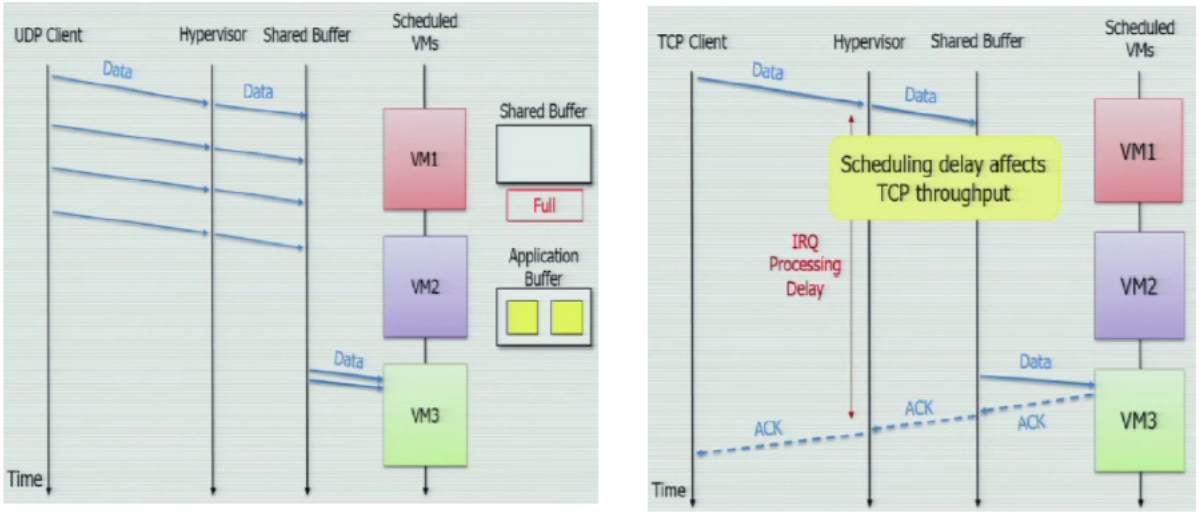


Figure 25: Challenges with shared vCPU cores. a) UDP packet receive b) TCP packet receive (Source: [28])

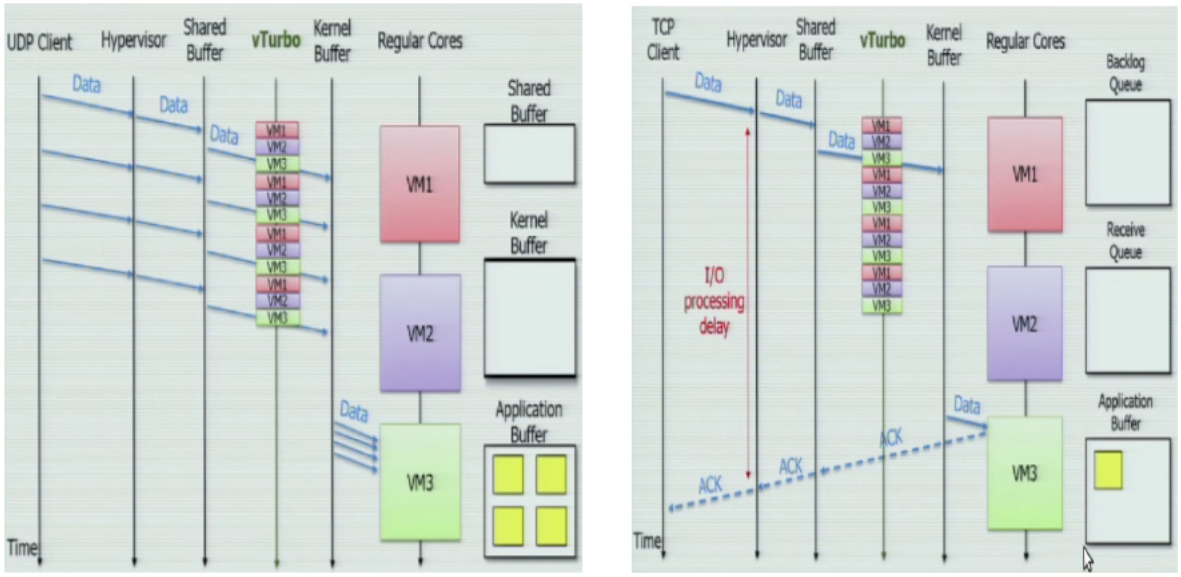


Figure 26: vTurbo solutions a) UDP packet receive b) TCP packet receive (with locked receive queue) (Source: [28])

running on turbo core. IRQ thread checks if the data segment is received in-order; if yes then it is marked as acknowledged and queued in the backlog queue; else falls back to the regular slow I/O process delivery technique. vTurbo requires hypervisor and guest kernel modifications to achieve the above.

Hypervisor modifications: We need to designate a set of cores as turbo cores when hypervisor initializes, and restricts turbo vCPUs to migrate on turbo cores as well as regular vCPUs to migrate on regular cores. We need to modify credit scheduler algorithm to encorporate turbo cores.

Guest OS modifications: Guest kernel buffer size needs to be tuned so that it is not full until regular

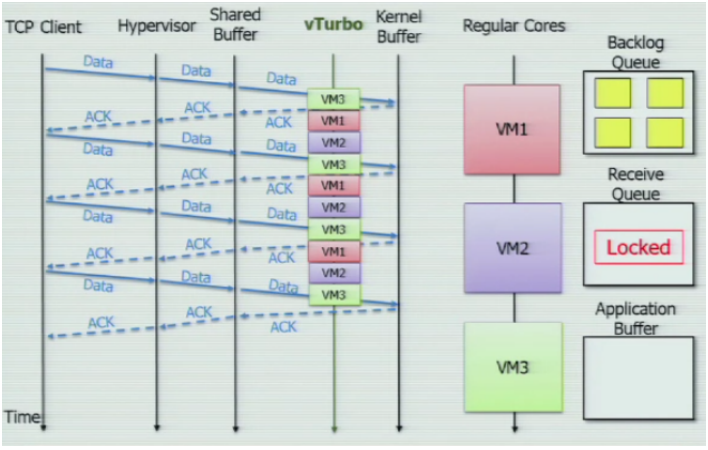


Figure 27: vTurbo Solution for TCP packet receive with TCP ACK generated by backlog queue (Source: [28])

core is scheduled. We also require modifications to guest TCP stack to enable ACK generation by IRQ context running on turbo core, if data has arrived in-order.
Limitations of this approach is data copy path is not optimized. End-to-end TCP semantics are not preserved, which could lead to inaccurate assumptions.

2.1.3.4 Packet Aggregation / Interrupt Coalescing

Some of the I/O optimization techniques is to optimize the I/O path; whereas others try to reduce the number of traversals over the I/O path. Packet aggregation or interrupt coalescing fall under the second category of optimization techniques. We can perform packet aggregation optimization on the transmit path using techniques like TSO; and also on the receive path using techniques like acknowledgement offload. VPE [29] uses polling technique to eliminate interrupts from critical I/O handling path.

TCP segmentation offload (TSO)

Authors of [5] propose TSO, wherein guest OS supports transmit processing of packet sizes larger than MTU (1500B). In absence of TSO support at the guest virtual interface , each 1500 byte guest packet transmit requires a page remap operation and one bridge forwarding operation. With TSO support at the virtual interface OS can potentially transfer 4096 bytesof data with one page remap operation, and would require a single packet forwarding operation over network bridge. TSO helps in reducing the per-byte processing overhead; and also reduces number of packets for same data. To implement TSO, NIC should have checksum offload and TSO feature. The downside of this approach is latency-sensitive applications may suffer from high delay and jitter.

Receive aggregation and Acknowledgement offload

Authors of [6] propose receive aggregation and acknowledgement offload tecniques to optimize the receive I/O path. Packets received from NIC are preprocessed, i.e., aggregated before received by the network stack. Packet aggregation is done only for the in-oder and error-free received packets that belong to same connection. Aggregation is not performed on ACK packets. TCP stack modifications are required as TCP depends on the number of packets received and ACK number received. This dependency is to ensure accurate congestion control mechanisms. The limitation again is higher delays for LSVMs.

In vSnoop [27], Dom0 acknowledges TCP packets on behalf of the guests whenever it is safe to do so. vSnoop is a program module working in Dom0. vSnoop acknowledges only in-order packets, and allows the guest to handle out-of-order packets. vSnoop acknowledges only if the buffer between the driver domain and guest VM is not full. This is done so that all packets acknowledged reach the guest VM, and semantics are preserved.
In vFlood [25], TCP sender floods Dom0 when required and offloads TCP congestion control to Dom0. vFlood resides in VM; shuts off default congestion control and floods the driver domain as fast as allowed. It requires a congestion control module in driver domain on behalf of VM. It also requires a buffer management module in driver domain that controls the flooding of packets so that buffer space for flooded packets is used fairly accross all connections and VMs.

Summary

All optimization solutions that we have seen, aim at reducing the CPU consumption and/or speedup the network I/O path. vBalance and vSlicer were able to reduce the latency to about 180 times, and 80% respectively, as compared to the traditional paravirtual model; whereas ELVIS reduces it to about 1.2 to 3 times. vBalance and vSlicer were able to improve the TCP throughput by 86%, and 200% respectively, as compared to the traditional paravirtual model; whereas ELVIS improves it by about 6% to 300%.

vBalance and vSlicer could be implemented for latency sensitive workloads, whereas ELVIS could be implemented for workloads that require high throughput.

2.2 Optimizations for Emulated device model

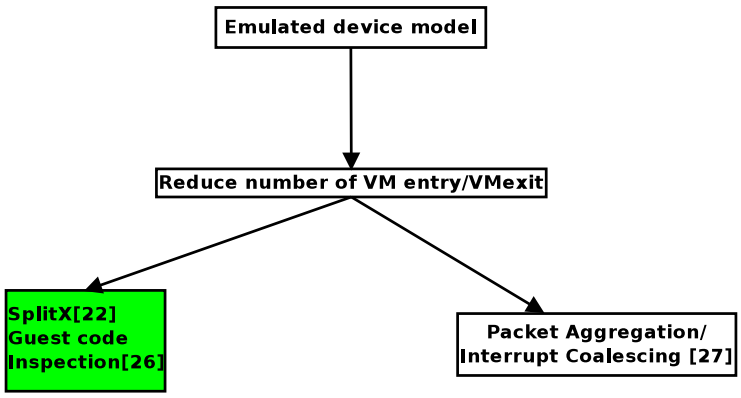


Figure 28: Classification of Network I/O Optimization Approaches for VMM using Emulated device model

Emulated device model is not explored much by the researchers. The basic source of overhead in this model is large number of VM entry and VM exit; these cause many context switches, and pollute the cache. Primary goal of optimization techniques with this model would be to reduce the number of VM entry/exit.

2.2.1 Reduce the number of VM entry/VM exit

SplitX [19] proposes dedicated core for guests and hypervisor. It aims at eliminating the VM entry and VM exit; which in-turn reduces context switches, and hence avoids performance losses due to

cache pollution. Whenever hypervisor intervention is required, guest communicates to the hypervisor through inter-core communication (hardware specific). This communication is asynchronous in nature. The hypervisor is notified about a new event using approaches like interrupt, cache-line monitoring ar polling. Hypervisor might use fast/ slow action depending on type of request. SplitX requires a x86 architectural modification for InterProcessor interrupts(IPI), which is currently available as; Intel, posted interrupts or AMD, doorbell interrupts.

Request notifications are sent by guest to the hypervisor using signals. Hypervisor executes a software handler to handle guest‘s request. The guest core does not execute the software handler, the core handles the signal. If the core finds that the guest is stalled at a synchronization point, it is resumed. If the guest is running, the next synchronization point is canceled.

Request completion notifications are sent to the guest by the hypervisor using signals. This notification does not cause an exit, and also carries information, like guest register values.

Hypervisor needs to be able to send resource management instructions to the guest cores through some inter-core bus communication .

Advantages of SplitX are as follows:

1. Guest executes without any request/reply notification exits.
2. Guest and hypervisor cache is not polluted due to dedicated cores.
3. SplitX could benefit by replacing the hypervisor core by a simpler, cheaper, and less power-hungry core, whereas use a full-featured core for guest.

Interrupts arrive only at hypervisor's core via exitless Inter-Core communication, but hypervisor is still involved in the I/O path

Authors of [22] propose improvements to the *emulated device model*, by reducing the number of context switches. VMM inspects guest code, and determines the back-to-back pair of instructions that both exit. Now, the VMM exits only once for a pair of instructions, improving the performance by 50%. It also generalizes pair of instructions to loops, and other control flow instructions. A binary translator is used to generate, and cache translations for handling exits. This translation cost occurs only once. The speedup achieved by this technique is around 152%.

The drawback of this scheme is analysis of guest code, whenever it is modified. This might not be a major drawback, but the problem is that even after optimizations, the performance of emulated model is poor than paravirtualized, and Direct I/O models.

2.2.2 Scheduling techniques

As in paravirtual model, in emulated device model researchers have proposed scheduling techniques like Polling and Packet Aggregation. They help reduce the number of traversals over the I/O path; implies reduction in number of exits, reduction in number of context switches, and avoid cache pollution.
[23] and many more propose packet aggregation and interrupt coalescing to improve performance.

Summary

All optimization solutions that we have seen, aim at reducing the number of context switches and/or speedup the network I/O path. SplitX and Batching techniques were able to improve the network throughput by 64%, and 200% respectively, as compared to the traditional emulated model. But the

traditional emulated model itself is capable of achieving about 10% to 15% throughput as compared to the native linux.

Enough literature is not available in this area. The reason could be that, this model inherently provides very poor performance, hence researchers are not interested in optimizing it.

2.3 Optimizations for Hardware Assisted device model

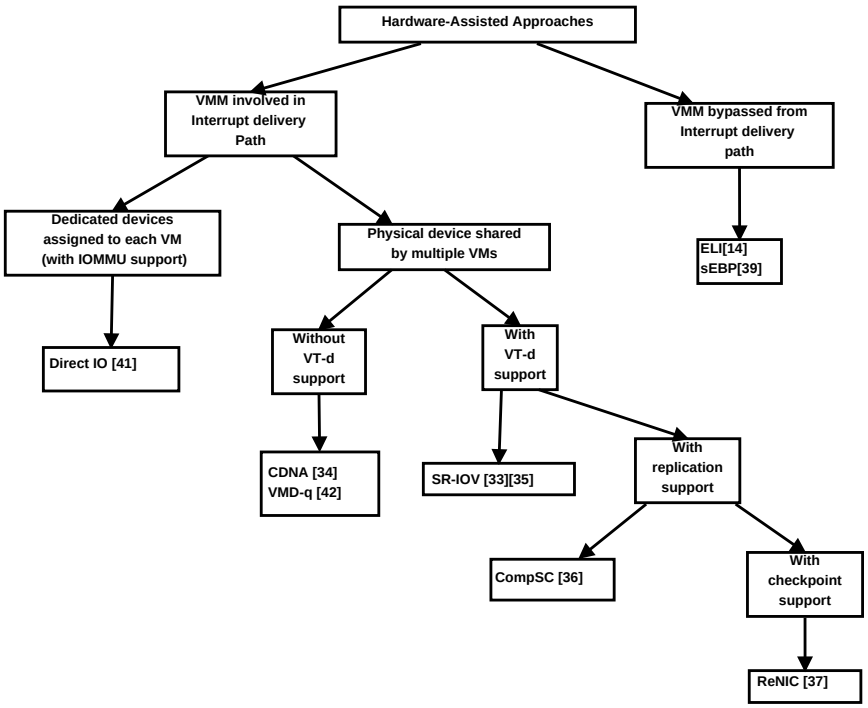


Figure 29: Classification of Network I/O Optimization Approaches for VMM using Direct I/O model

Authors of [13] [29] [37] have shown through results that Direct I/O assignment techniques eliminate significant overhead as compared to software I/O virtualization techniques. Figure shows that multiple VMs share the same physical NIC, and the multiplexing/demultiplexing happens via the software switch. The advantage of this approach is it does not require special hardware support, but at the cost of high CPU overheads. Also VMs cannot access the data directly. To enable direct communication between the physical NIC and the application, it was necessary to copy the data packets to the guest through DMA. But DMA requires physical addresses for data transfer, so there was a requirement for translation of virtual address to physical address securely. This operation is performed by IOMMU (I/O memory management unit).

2.3.1 Directed I/O (VT-d)

Intel virtualization technology for Directed I/O (Vt-d) [45] is the hardware extension (IOMMU) to support VMM bypass, and allow guests to directly talk to the hardware. In directed I/O a dedicated NIC is assigned to each guest. Intel VT-d ensures security, reliability, and improves virtualization performance. The downside of Directed I/O is that, it cannot scale well.

2.3.2 Solutions without VT-d support

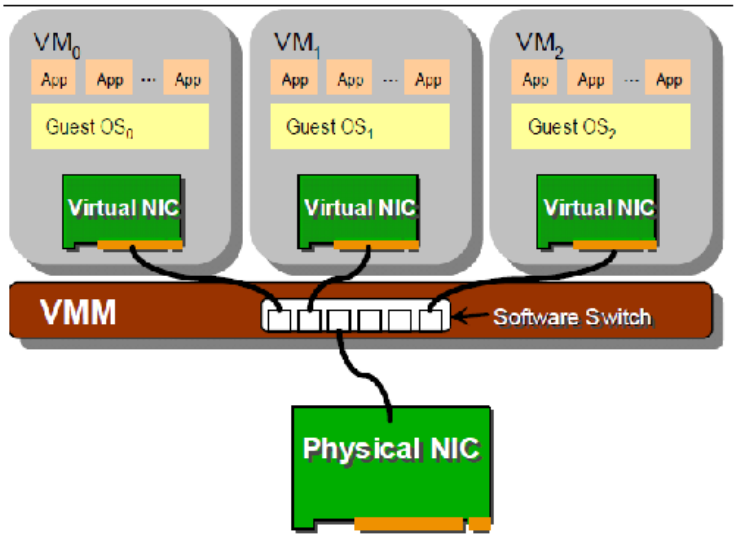


Figure 30: Software I/O virtualization

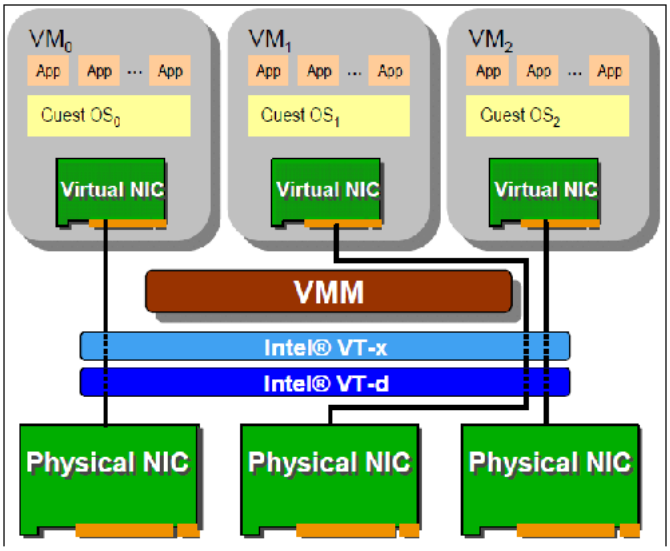


Figure 31: Intel VT-d technology

2.3.2.1 CDNA CDNA [30] uses software and hardware modifications to improve virtualized I/O performance. It divides virtualized I/O access functions into three parts; Traffic multiplexing, Interrupt delivery, and Memory protection. It distributes these functions among hardware and software. Traffic Multiplexing is performed on NIC; whereas interrupt delivery and memory protection is performed by VMM with NIC support. CDNA NIC supports multiple context in hardware; can act as independent physical NIC, or can be controlled by separate device driver instance. Guest can directly transmit/receive network traffic using its private context.
Software traffic multiplexing is eliminated, as the hypervisor maps I/O locations for guests private contexts mailboxes into guest address space. Hypervisor is still involved in control functions, guaranteeing memory protection and guest virtual interrupt delivery.

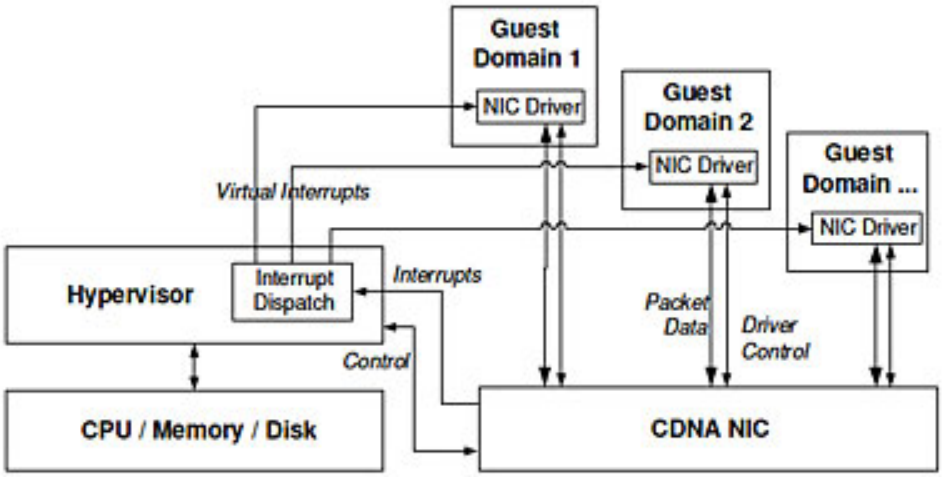


Figure 32: CDNA architecture (Source: [30])

NIC keeps track of contexts that have changed since last physical interrupt, encodes in a interrupt bit vector, DMAs it to the hypervisor address space. After this transfer, NIC raises a physical interrupt, and hypervisor schedules a virtual interrupt for the guest.
To maintain isolation between guests; CDNA validates and protects all DMA descriptors and ensures that guest maintains ownership of physical pages that are sources or targets of outstanding DMA accesses. DMA descriptors are enqueued by the guest through the hypervisor. The advantage over directed I/O approach is that now the physical NIC can be shared by multiple VMs; hence scalability is better. The downside of this approach is still interrupt delivery and memory protection is handled by software (VMM); which still remains a bottleneck. CDNA also requires a specfic hardware to function.

2.3.2.2 VMD-q Similar to CDNA, VMD-q (Virtual machine device queue) [38] also offloads packet classification to the network adaptor. VMM has to intervene to ensure memory protection and address translation. VMM has to copy the packet to VM; this turns out to be a bottleneck. VMD-q technology is provided by Intel, and is faster as compared to vanilla Xen/KVM due to some function offloading.

2.3.3 VT-d and sharable device support

2.3.3.1 SR-IOV Single Root - I/O Virtualization and sharing (SR-IOV) [33] is fast I/O virtualization standard in PCI Express devices. SR-IOV is a PCI-SIG specification that defines a standard for creating natively shared devices. In SR-IOV, packet multiplexing, address translation, and memory protection is performed by the hardware. To perform address translation and memory protection securely, SR-IOV uses Intel VT-d technology, which provides hardware functioning of IOMMU. Hypervisor is completely eliminated from the latency-sensitive I/O path. CPU is no longer involved in copying data to, and from the VM.
SR-IOV capable device is a PCIe device, that can create multiple virtual functions (VFs). A PCIe device is a collection of one or more functions. An SR-IOV capable device has one or more Physical functions(PFs); and each PF is a standard PCIe function associated with multiple Virtual functions (VFs). Each VF acts as a light-weight PCIe function, that is configured and managed by PFs. SR-

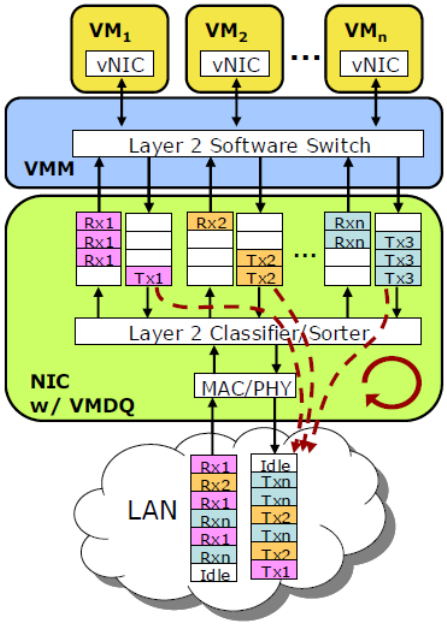


Figure 33: VMD-q architecture

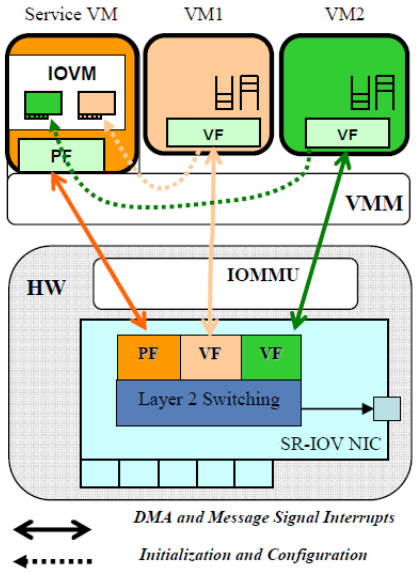


Figure 34: SR-IOV virtualization architecture (Source [29])

IOV comprises of three components; PF driver, VF driver, and SR-IOV manager (IOVM).
The PF driver has access to all PF resources. It manages and configures VFs. At startup, it sets number of VFs, enables/disables VFs, sets device configurations like MAC address and VLAn settings for a NIC, configures layer 2 switching.
The VF driver executes on the guest OS, and can access its VF directly (without VMM involvement). VF needs to duplicate resources such as DMA descriptors, that are performance-critical; whereas other resources are emulated by IOVM and PF driver.

IOVM provides a virtual full configuration view of each VF to the guest OS, so that the guest configure VF as a physical device. IOVM helps dynamic addition of VFs to the host, which are then assigned to the guest OS. Once guest discovers the assigned VF, it can initialize and configure it as any physical device. PF and VF driver communicate with each other using hardware-based producer/consumer technique. Producer writes a message into the mailbox, and rings the doorbell. The consumer consumes the message, and notifies the producer by setting a bit in shared register. Figure

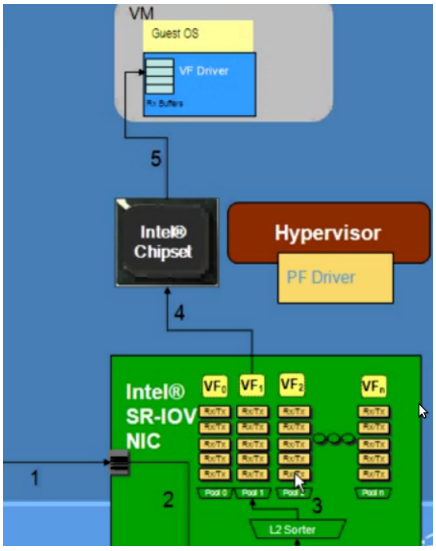


Figure 35: SR-IOV working (Source [29])

35 shows the working of SR-IOV

1. A packet arrives at the SR-IOV NIC
2. It is sent to L2 sorter
3. It is sorted based upon MAC/VLAN, and placed into corresponding VF queue
4. DMA action initiated to transfer the packet to the guest address space. This is achieved securely, and without VMM intervention using Intel VT-d technology.

SR-IOV achieves high throughput (native), and is also scalable. The problem with SR-IOV is high CPU consumption. This is due to VMM intervention for guest interrupt delivery. VMM captures the physical interruptfrom VF, maps to guest virtual interrupt, and injects it. VMM needs to emulate virtual Local APIC for HVM guest and event channel for PVM guest. This overhead is high, as interrupt frequency is about 70K per second per queue. The CPU utilization for a 1 Gbps SR-IOV NIC is approximately 499% [29]. Authors suggest multiple optimizations to reduce the CPU utilization from about 499% to 200-227% [29]. Another problem with SR-IOV is that, it is extremely difficult to replicate the hardware state of NIC due to high frequency and non-deterministic nature of incoming packets.
Most intuitive solution to this problem is; dynamic switching between direct accessed VF at runtime, and emulated VF at migration time. This would work, but requires development of per-device hypervisor specific model, which a lot of software engineering effort.

2.3.4 Replication support for SR-IOV capable device

2.3.4.1 CompSC Challenges associated with Replication/Live Migration are: (1)The state of a device needs to be efficiently read and written to support device replication. (2) The dirty memory

written by the device DMA needs to be efficient, and tracked for lazy memory state transmission. Requirements of efficient VM Replication:

1. Device state needs to be efficiently read/written; to support device-state replication; at high frequencies
2. The device needs to efficiently buffer the output packets; until they can be released after successful checkpoints
3. The dirty memory written by DMA needs to be efficient; and tracked for lazy memory state transmission

CompSC [32] attacks the live-migration problem with SR-IOV capable devices. It tries to solve the problem of hardware state restoration. CompSC proposes a record/replay mechanism, in which every hardware access is recorded (Recording stage), and is replayed at the destination (Replay stage).
This record/replay for high frequency I/O is practically not feasible. CompSC suggested few optimizations like; (1) Record last register write, when writing has no side effect. (2) Define operation sets (op set), the op sets is Critical Section.
Op sets in Intel 82576/82599 NIC are initializing operations, sending operations, receiving operations,other remaining op states include only uninitialized, up, down. In this kind of set up, only the latest operations on each setting register and whether or not the interface is up need to be tracked. There is specific design required for statistic registers. For example, a dropped packet counter has

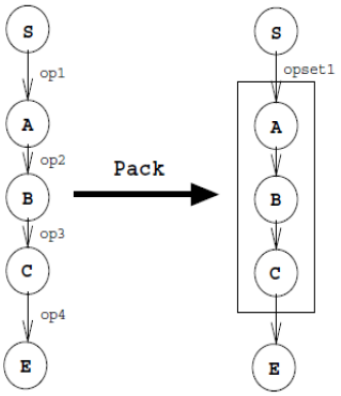


Figure 36: Opsets (Source [32])

value 'm' before migration, is reset at migration initiation, and has value 'n' after end of migration. Then the actual value to be set after migration is 'm+n'. To replicate I/O state, it is necessary to track

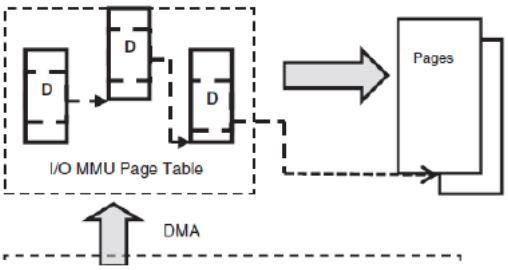


Figure 37: DMA dirty page tracking (Source [32])

the pages modified during device DMA operation. This is not supported with current IOMMU. The

solution is dummy write the page for which DMA operation is performed, after the DMA operation. CompSC migration involves the following stages: Pre-migration, Reservation, Iterative pre-copy, Stop

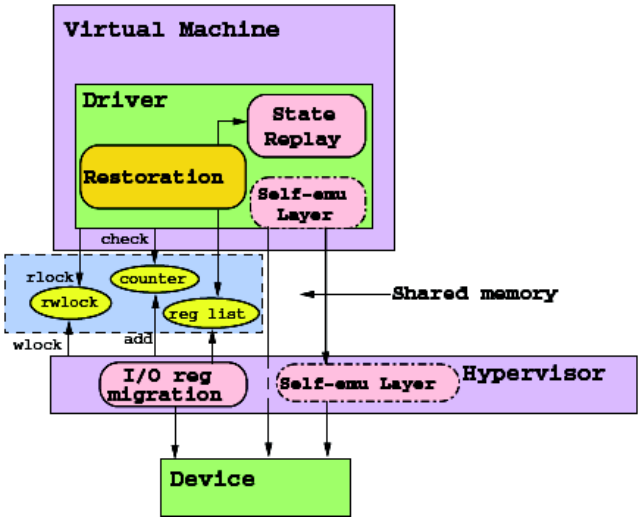


Figure 38: CompSC architecture (Source [32])

and copy, Commitment, and Activation. This solution does not have checkpoint support.

2.3.5 Replication and Checkpoint support for SR-IOV capable device

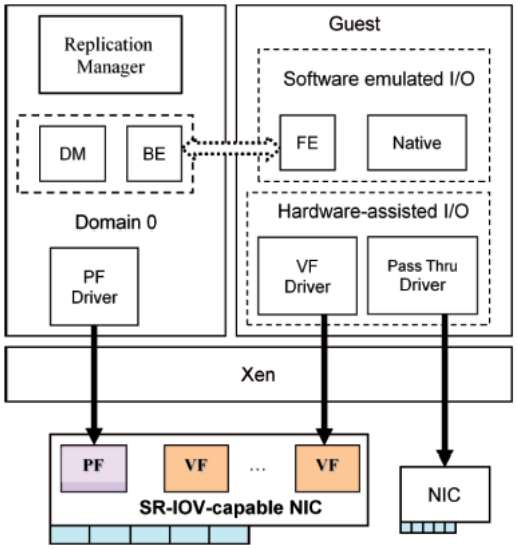


Figure 39: ReNIC architecture (Source [33])

2.3.5.1 ReNIC ReNIC [33] ensures live-migration of SR-IOV capable devices with consistent register state post-migration. It also incorporates Remus checkpoint like solution.
It proposes two states; Working mode and clone mode, where Working mode is further divided into buffering and releasing mode. ReNIC supports atomic read/write of a VF state (Clone mode). It

supports outbound packet buffering. (buffering, releasing mode). It supports tracking of DMA dirty pages (extends existing IOMMU architecture)

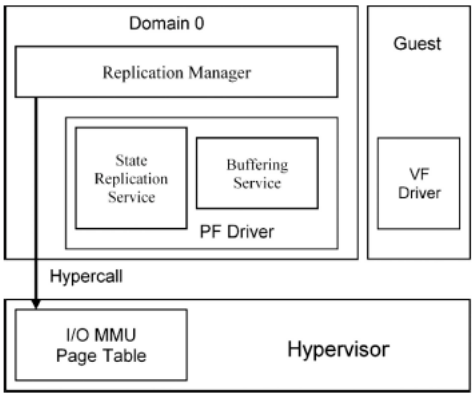


Figure 40: ReNIC hypervisor extensions (Source [33])

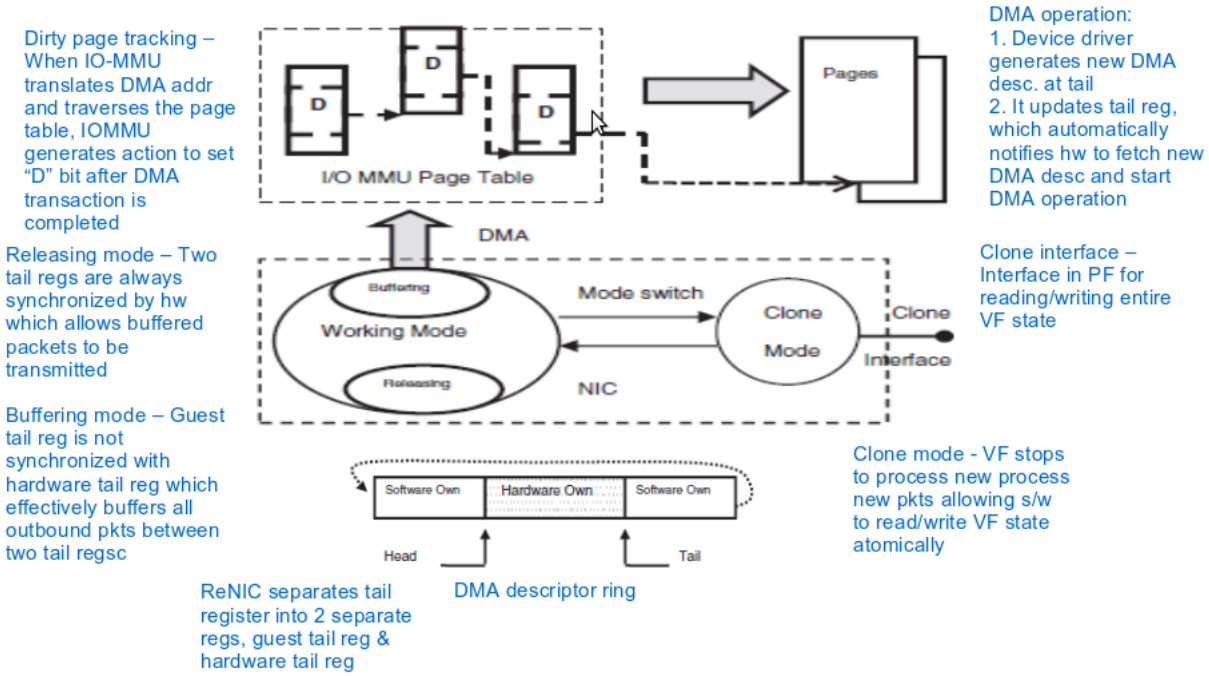


Figure 41: ReNIC details (Source [33])

Address Translation Service(ATS) in IOMMU provides I/O TLB to bypass page table traversal at run-time to improve performance. ReNIC needs to turn OFF this service. But this increases latency; leading to poor performance.
ReNIC's solution is to configure I/O MMU to temporarily disable ATS translation request for the in-migration VF, flush translation cache in VF, when dirty page tracking is turned ON.

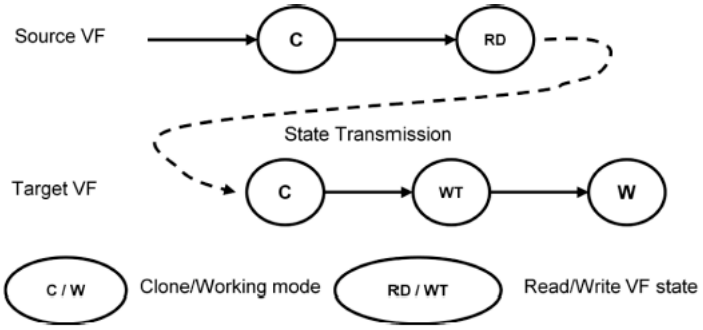


Figure 42: ReNIC VF state replication (Source [33])

2.3.6 VMM bypass for Interrupt Delivery path

ELI [14] improves I/O throughput and latency of unmodified, untrusted guests to reach near native performance. It eliminates the host from interrupt-handling path. Interrupts generated by assigned

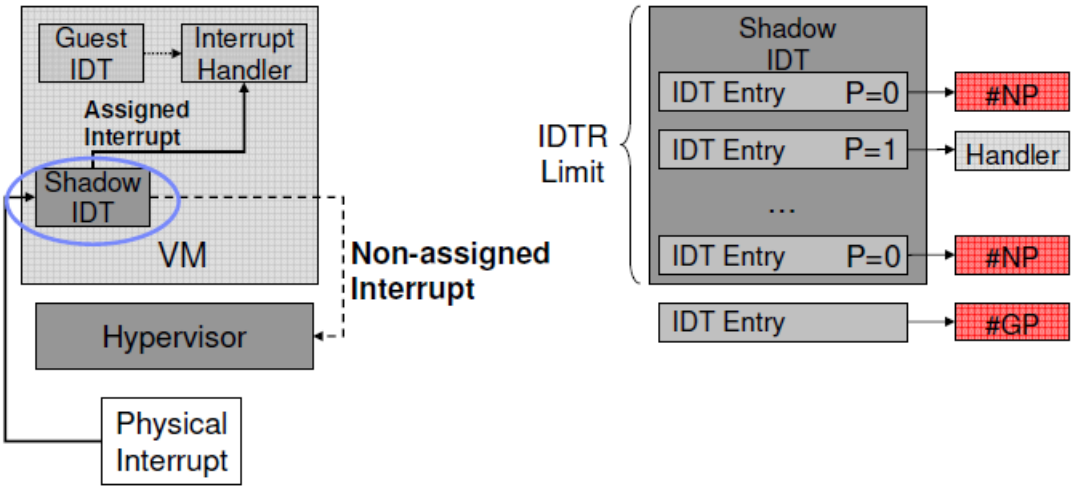


Figure 43: ELI working (Source: [14])

devices for guests are received directly by guests; while other exit to the host. Guest can signal interrupt completion without causing an exit. Guest maintains its own IDT. For Exitless interrupt delivery, ELI runs guest in guest mode with an IDT prepared by the host called Shadow-IDT. Shadow IDT entries are copied from guest original IDT and are marked as present. Host configures shadow-IDT to deliver assigned interrupts directly to guests interrupt handler OR force an exit for non-assigned interrupts. This is done by adding Present/Not-present bit in IDT. Every entry marked NP is forced to NP exception, and exits from guest mode to host mode. Devices emulated by the host are handled using virtual interrupts injected by host to the guest, and these entries are marked NP. To deliver such interrupts through guest IDT handler, ELI enters in a special injection mode by configuring processor to cause an exit on any physical interrupt and running guest with original guest IDT.

For Exitless interrupt completion notification, guest notifies by writing EOI LAPIC register. ELI exposes x2APIC EOI register directly to the guest by configuring MSR bitmap to not cause an exit when guest writes to EOI register. During injection mode, host temporarily traps accesses to EOI

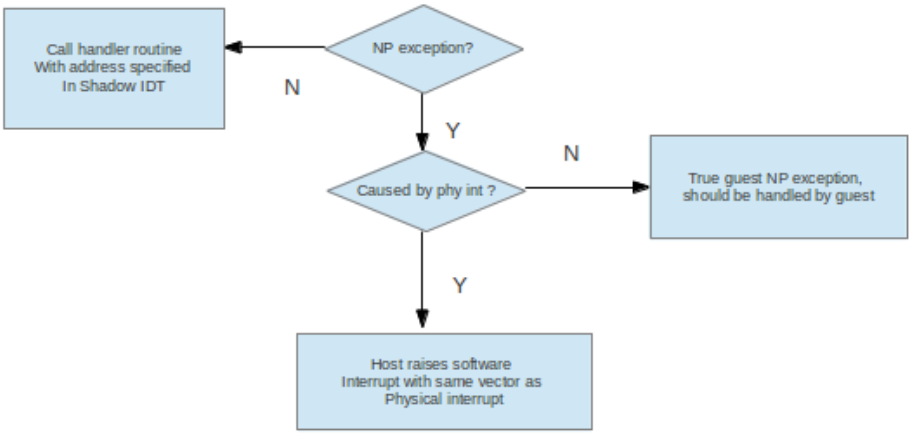


Figure 44: NP exception handling

register, as completion notification should go to host for emulation.

sEBP [36] eliminates interrupts from the critical I/O path using polling technique. Polling interval is decided on the basis of events generated at both guest and host level. It comprises of an event collector and event manager. Event manager comprises of Rate controller and polling module. Rate controller uses adaptive technique to decide the rate at which polling should be done. Polling module decides the technique of communication between guest and host.

Summary

VMM bypass direct I/O seems to be the next generation I/O virtualization technique. This model is capable of gaining maximum throughput, and lowest latency, by eliminating VMM from latency sensitive I/O packet path. It is possible to achieve wire-speed performance with SR-IOV technique. But the CPU consumption increases, while improving the throughput. Techniques like ELI, and sEBP eliminate VMM from interrupt delivery path too.

The major drawbacks of this model are –

1. There is not enough support for scalability.
2. Live Migration process requires some architectural modifications.
3. Since VMM is eliminated from the I/O path, appropriate resource control is a challenge.

There is enough scope in the *resource control*, and *live migration* domain in the direct I/O model.

2.4 Comparison of Network I/O Optimization Approaches

Table 8 provides significant *Sources of Overhead for I/O access* for different virtualization techniques. Tables 9, 10, and 11 compare major techniques in Paravirtualized, Emulated, and Direct device assignment model.

Table 8 would help us in choosing the target problem to attack, so as to optimize a particular virtualization model.

Table 8: Sources of Overhead for Different VMMs Without Optimizations

Sources of Overhead	Xen	KVM-HVM	KVM-virtio	Direct I/O approach
Additional Packet Copy on receive side	✓	✗	✓	✗
Netfilter routine	✓	✓	✓	✓
Bridge Routine	✓	✗	✗	✗
Page pinning	✓	✗	✓	✗
Grant operations	✓	✗	✓	✗
VM entry/exit	✗	✓	✗	✗
QEMU Emulation	✗	✓	✗	✗
TAP transaction	✗	✓	✓	✗
Virtual Interrupt delivery	✓	✓	✓	✓

3 Open Problems

The proposed paravirtualized model optimization solutions, aim at reducing, or eliminating the hypervisor intervention from the latency sensitive I/O path. The proposed emulated model optimization solutions, aim at reducing, eliminating, or reducing the cost of the context switching overhead, from the latency sensitive I/O path. The goal of these approaches is to speed up the virtualized I/O path.

The literature has covered almost every corner, leaving no gaps in the *Network I/O Path Optimization* domain. There are some open problems that are unveiled, after implementing optimizations, like Live-migration challenge with Direct I/O model. There are other open problems that existed even without optimizations, like VM application QoS satisfaction, Dynamic VM network policy setup.

3.1 Dynamic Setup for VM Network I/O policies

It is necessary to efficiently, and accurately manage the system resources. The applications running on VMs get as much of the resource share as required by the applications running on it. There are many approaches proposed by researchers, that try to get hints from the VM's applications to configure the hypervisor scheduler. But yet no approach has been incorporated into the hypervisors. The reason could be that , the solutions are'nt generic, or they take up too much of resources, which eventually reduces the share of the VMs.

A generic solution to set up network I/O policies for VMs dynamically, and without affecting the performance of running VMs is a open problem. There could be two possible approaches:

1. White-box approach – Allowing VM to specify the parameter values for a policy. With this approach, the service user would pay only for the kind of service, that he requests. For e.g., cost of best-effort service would be comparatively lower than that of a deterministic service. This is not a generic approach, as it requires some communication between the VM and the VMM.
2. Black-box Approach – Allow a controller to continuously monitor the traffic coming from the VMs, learn it, and dynamically set policies such that application QoS is not lost. We propose an approach towards dynamic configuration of a hypervisor scheduler, by taking hints from incoming/outgoing packet headers, using properties like, port numbers, and use some machine learning algorithm to predict the resource requirement of the VM application.

3.2 Offload Netfilter Functions

[4] has shown through their experiments, that bridge/netfilter routines have 15% of transmit overhead and 19% of receive overhead. Every organization has some policies that the network users have to abide to. In current Xen optimization approaches, net-filter is turned OFF to reduce virtualized

Table 9: Comparison of Paravirtualized I/O Optimization Techniques

	vBalance	vSlicer	vTurbo	ELVIS
Approaches				
Migrate interrupts from pre-empted vCPU to running vCPU	✓			
Balance interrupt load on CPU	✓			
Increase Scheduling Frequency for LSVMs		✓time_slice=5ms	✓time_slice=0.1ms	
Requires Hypervisor Modifications	✓	✓	✓	✓
Requires Guest Modifications	✓		✓	✓
Exitless Request Notifications				✓
Exitless Completion Notifications				✓
Performance				
Latency Reduction wrt Traditional Paravirtual I/O (RTT)	180x	80%	-	1.2x to 3x
Throughput Improvement wrt Traditional Paravirtual I/O	TCP: 86% UDP: 102%	TCP: 200%, but cannot achieve wire speed	Disk_Write: 75% Disk_Read: 26% TCP: 63 to 200% UDP: Wire speed	6 to 200%
Limitations	Increased number of context switches by almost 69% for web_server workload	1. Classification of VMs into LSVMs and non-LSVMs is to be done dynamically 2. Reduces Scheduling latency, does not eliminate it	1. Data copy path is not optimized 2. End to End TCP semantics are not preserved	Use of dedicated vCPUs for each core is suggested (to avoid performance degradation)

network I/O overhead. Therefore, offloading network level rules, policies and traffic rate limits from VMM to hardware is still an open problem. A solution to this would definitely help in improving virtualized network I/O speeds along with netfilter functions.

3.3 Live-migration Challenge in Direct I/O model

After the literature survey, it can be observed that Direct I/O model solutions like SR-IOV provide best virtualized I/O access performance. The basic drawback of this model, is limited scalabilty, and challenging live-migration. Solutions like CompSC, and ReNIC propose live-migration for SR-IOV,

Table 10: Comparison of Emulated I/O Optimization Techniques

	SplitX	Batching/Interrupt Moderation
Approaches	1. Eliminate VM entry/VM exit 2. Dedicated Cores for Guest/Hypervisor 3. Inter-core communication for Guest-VMM communication 4. Uses IPI for event notification	1. Implements Interrupt Moderation 2. Implements Send Combining
Performance		
Throughput Improvement wrt Traditional Emulated I/O	Netperf: 64%	UDP_transmit: 200% UDP_receive: Depends on incoming traffic characteristics
Limitations	VMM is involved in the interrupt delivery path	Batching introduces additional packet processing delays, and increases the response time

but they require major architectural changes. Live-migration solution without significant architectural modifications, still remains an open problem.

3.4 Scalability Challenge in Direct I/O model

In VMM Bypass Direct I/O technique, a separate NIC is assigned to each VM. There is a restriction on the number of NICs installed on a machine, hence the restriction on the number of VMs that can

In SR-IOV solution, the number of VF's are fixed for a SR-IOV NIC. Each VM is assigned a fixed VF, so the maximum number of VMs are restricted by the number of VFs supported by the SR-IOV NIC. This is another open problem.

3.5 Proposed Approach

[47] has developed a mechanism for Network endpoint reconfiguration for Xen hypervisor. The proposed design is shown in Figure 45. 45 has implemented the technique by which VMM could allow VMs to dynamically switch over from SR-IOV VFs to PV vNICs, and vice-versa using a single interface. VM has a common interface to connect to the VF, and the netback. These connections permanently exist, but only one connection is active at a time. When the VM is initialized, the default active connection is PV vNIC. The Controller module (thread) in the VMM runs continously as a background process. Netback can communicate with the controller module, to configure VM endpoint to VF, or vNIC dynamically. This does not require a VM restart.

Using the above design, multiple use cases can be proposed:

1. Scalability support for Direct I/O model – Say, SR-IOV supports 'm' VMs, and we wish to provision 'n' VMs, where $n > m$, then 'm' VMs could be assigned VFs with deterministic service; whereas 'n-m' VMs could be assigned PV vNICs with best effort service.

Table 11: Comparison of Direct I/O Optimization Techniques

	CDNA	VMD-q	SR-IOV	CompSC	ReNIC	ELI	sEBP
VT-d support	✗	✗	✓	✓	✓	✓	✓
Multiplexing / Demultiplexing of Packets by Hardware	✓	✓	✓	✓	✓	✓	✓
Memory Protection by Hardware	✗	✗	✓	✓	✓	✓	✓
Resource control in Hardware	✗	✗	✗	✗	✗	✗	✗
VM Interrupt Delivery by Hardware	✗	✗	✗	✗	✗	✓	✗
VMM bypassed for Interrupt Delivery/Completion Notifications	✗	✗	✗	✗	✗	✓	✓
Is Live Migration a Challenge?	No	No	Yes	Somewhat	No	-	-
CPU Consumption wrt Native	↓ 51%	-	HVM:1VM:↑ 17% HVM:7VMs: ↑ 30%	-	-	-	-
Throughput Improvement wrt Native	Tx: 200Mbps Rx: 750Mbps	-	Wire-speed	-	-	49% to 66%	WebBench: 59%

2. Network Policy layer – There can be a policy layer above the substrate, that could feed the controller with commands, to switch a VM from VF to vNIC, and vice-versa. The details on this were discussed in section 3.1.

3. Live-Migration support for Direct I/O model – The problems faced during live-migration are mentioned in section 2.3.3.1. We can attack this problem by switching to PV mode during migration, and switching back to Direct I/O mode after migration. Using the above design, during live-migration, the controller can switch the VM connection from VF to vNIC. At the destination, after end of migration process, the controller can switch the VM connection back from vNIC to VF.

4 Future Work

In this report, virtualized I/O performance is studied for communication from VM on a PM to the outside world. But it is also essential to study, InterVM-IntraHost communication techniques and their optimizations as; (1) Many Placement algorithms try to keep VMs that communicate frequently on same PM. For example, a Web server may communicate with a Application Server, or a Database Server more frequently. (2) With Software-defined networking, entire organizational network is treated as a single virtual disk. It is necessary to know the techniques by which VMs communicate with each other in this scenario.

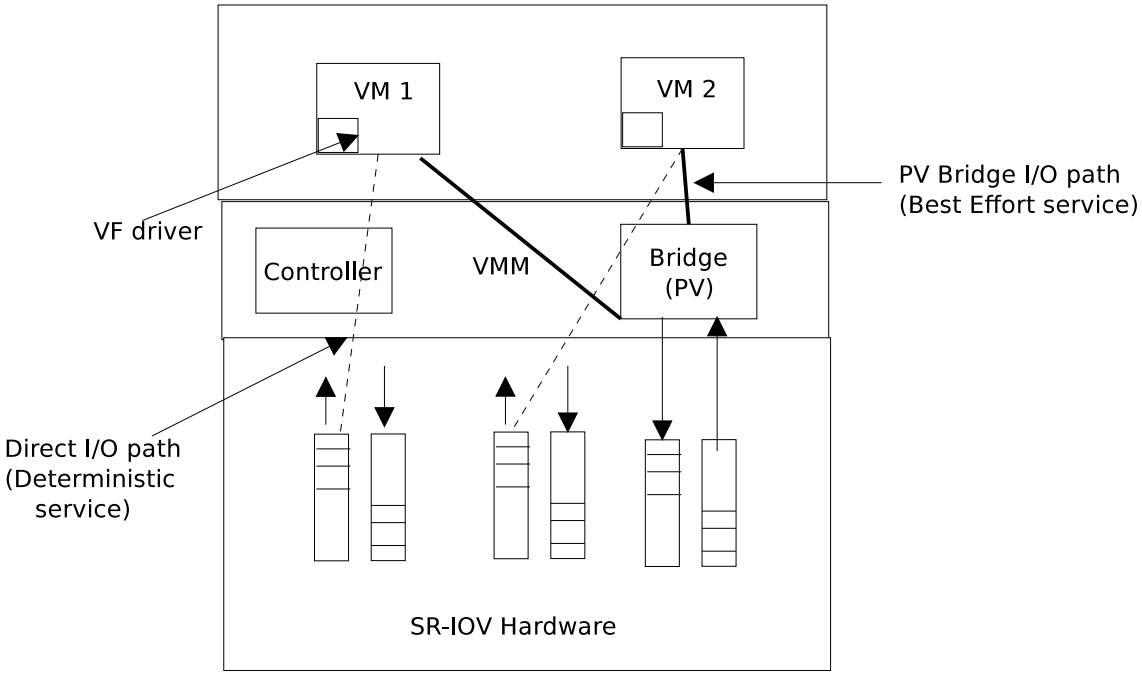


Figure 45: Proposed Approach (Substrate Design by [47])

5 Conclusion

In this report, we have analyzed various sources of overhead in Xen and KVM hypervisor. We contribute by providing a classification on virtualized I/O communication approaches. A detailed survey performed on the Optimization approaches for virtualized I/O communication. We also present a comparison on the optimization techniques.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, and Andrew Wareld. Xen and the Art of Virtualization. In SOSP, 2003.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In OLS, 2007.

[3] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. In ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel archive, Volume 42 Issue 5, July 2008, Pages 95-103

[4] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In First ACM/USENIX Conference on Virtual Execution Environments (VEE05), Chicago, USA, June, 2005.

[5] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In USENIX Annual Technical Conference, Boston, MA, June, 2006.

[6] Aravind Menon and Willy Zwaenepoel. Optimizing TCP Receive Performance. In USENIX Annual Technical Conference, 2008.

[7] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. TwinDrivers: Automatic Derivation of Fast and Safe Hypervisor Drivers from Guest OS Drivers. In ASPLOS, 2009.

[8] Padma Apparao, Srihari Makineni, and Don Newell. Characterization of network processing overheads in Xen. In VTDC, 2006.

[9] Fernando Laudares Camargos, Gabriel Girard, and Benoit des Ligneris. Virtualization of Linux servers. In Ottawa Linux Symposium, 2008.

[10] J. Renato Santos, G. (John) Janakiraman, and Yoshio Turner. Xen network I/O performance analysis and opportunities for improvement. In Xen summit, 2007.

[11] Jose Renato Santos, Yoshio Turner, G.(John) Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In USENIX, 2008.

[12] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10 Gb/s using Safe and Transparent Network Interface Virtualization. In VEE 2009

[13] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High Performance VMM-Bypass I/O in virtual machines. In USENIX, 2006.

[14] Abel Gordon, Nadav Amit, Nadav HarEl, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. In ASPLOS, 2012.

[15] Nadav HarEl, Abel Gordon, and Alex Landau. Efficient and Scalable Paravirtual I/O System, a.k.a ELVIS. In ATC, 2013.

[16] Abel Gordon, Nadav HarEl, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. Towards Exitless and Efficient Paravirtual I/O. In SYSTOR, 2012.

[17] HaiBing Guan, YaoZu Dong, RuHui Ma, Dongxiao Xu, Yang Zhang, and Jian Li. Performance Enhancement for Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive-Side Scaling. In TDPS 2012.

[18] Guangdeng Liao, Danhua Guo, Laxmi Bhuyan, and Steve R King. Software Techniques to Improve Virtualized I/O Performance on Multi-Core Systems. In ANCS, 2008.

[19] Alex Landau, Muli Ben-Yehuday, and Abel Gordon. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In WIOV, 2011.

[20] Jiuxing Liu, Bulent Abali. Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization. In ICS, 2009.

[21] Manel Bourguiba, Kamel Haddadou, and Guy Pujolle. Packet aggregation based network I/O virtualization for cloud computing. In Computer Communications 35 (2012) 309319.

[22] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software Techniques for Avoiding Hardware Virtualization Exits. In ATC, 2012.

[23] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. Speeding Up Packet I/O in Virtual Machines. In ANCS, 2013.

[24] Luwei Cheng and Cho-Li Wang. vBalance: Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines. In SOCC 2012.

[25] Sahan Gamage, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds (a.k.a vFlood). In SOCC, 2011.

[26] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing. In HPDC, 2012.

[27] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella and Dongyan Xu. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In SC, 2010 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.

[28] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In USENIX, 2013.

[29] Himanshu Raj and Karsten Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices In HPDC, 2007.

[30] Paul Willmann, Jeffreu Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In HPCA, 2007.

[31] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High Performance Network Virtualization with SRIOV. In HPCA, 2010.

[32] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. CompSC: Live Migration with Pass-through Devices. In VEE, 2012.

[33] Yaozu Dong, Yu Chen and Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication. In ACM Transactions on Architecture and Code Optimization (TACO) - HIPEAC Papers, 2012

[34] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleshwar K. Panda. High performance VMM-Bypass I/O in virtual machines. In USENIX ATC, 2006.

[35] Kun Tian, Yaozu Dong, Xiang Mi and , Haibing Guan. sEBP: Event Based Polling for Efficient I/O Virtualization. In CLUSTER 2012.

[36] HaiBing Guan, YaoZu Dong, RuHui Ma, Dongxiao Xu, Yang Zhang, and Jian Li. Performance Enhancement for Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive-Side Scaling. In IEEE Transactions on Parallel and Distributed Systems, VOL. 24, NO.6, June 2013

[37] LEVASSEUR, J., UHLIG, V., STOESS, J., AND G OTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In USENIX Symposium on Operating Systems Design & Implementation (OSDI) (2004).

[38] Intel VMDq Technology. In Notes on Software Design Support for Intel VMDQ Technology, March 2008,Revision 1.2

[39] Binbin Zhang, Xiaolin Wang, Rongfeng Lai,Liang Yang, Yingwei Luo , Zhenlin Wang, and Xiaoming Li. A Survey on I/O Virtualization and Optimization. In Chinagrid, 2010.

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

[41] Ryan Shea and Jiangchuan Liu. Network Interface Virtualization:Challenges and Solutions. In IEEE Network, September/October 2012.

[42] Shan Zeng and Qinfen Hao. Network I/O Path Analysis in the Kernel-based Virtual Machine Environment through Tracing. In International Conference on Information Science and Engineering (ICISE2009).

[43] M. Tim Jones. Virtio: An I/O virtualization framework for Linux. In IBM developerWorks, January 2010.

[44] Gal Motika and Shlomo Weiss. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. In Computer Standards & Interfaces 34 (2012) 3647.

[45] Abramson, D., Jackson, J.,Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., AND Wiegert, J. Intel virtualization technology for directed I/O. Intel Technology Journal 10, 3 (2006), 179192.

[46] http://en.wikipedia.org/wiki/Moore's_law

[47] http://www.cse.iitb.ac.in/synerg/lib/exe/fetch.php?media=public:students:kallol:kallol_stage1_report.pdf