VIRTUAL INTERRUPT HANDLING TO REDUCE CPU OVERHEAD IN I/O VIRTUALIZATION

Thesis

Submitted in partial fulfilment of the requirements of
BITS C421T Thesis

By

Nomchin Banga
ID No. 2009B4A7751P

Under the supervision of

Prof. Rahul Banerjee
(Chief, Software Development and Educational Technology Unit)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE (PILANI)

1st December, 2013

ACKNOWLEDGEMENT

I express my sincere gratitude to my advisor Prof. Rahul Banerjee for his guidance and support throughout my work. I am also grateful to the faculty of Department of Computer Science and Information Systems for mentoring me in certain areas of my research. I would like to thank the members of team Shakti for helping me throughout the learning process. I will also like to take this opportunity to appreciate the support and help that my family extended during the course of my work.

CERTIFICATE

This is to certify that the Thesis entitled, Virtual Interrupt Handling to Reduce CPU Overhead in I/O Virtualization, and submitted by Nomchin Banga, ID No. 2009B4A7751P in partial fulfilment of the requirement of BITS C421T Thesis embodies the work done by him/her under my supervision.

Signature of the Supervisor

Date: (Prof. Rahul Banerjee)

ABSTRACT

Thesis Title: Virtual Interrupt handling to Reduce CPU Overhead in I/O Virtualization
Supervisor: Professor Rahul Banerjee
Semester: First **Session**: 2013-14
Name of Student: Nomchin Banga **ID No.**: 2009B4A7751P

Abstract:

Cloud computing is a vital and one of the fastest growing models of IT in current scenario. Virtualization lies at the base of this computing model. However, there exist a few bottlenecks, including I/O virtualization. I/O devices and processor communicate with each other using hardware and software interrupts. Due to the numerous context switches between guest and host systems, virtual interrupts incur severe processing overheads. This thesis focused on reducing CPU overhead in interrupt processing by reducing the number of context switches. Basing the thesis on the architecture of Linux kernel integrated hypervisor KVM, a technique called coalescing has been applied to increase the efficiency of handling the interrupt generated due to network I/O.

Table of Contents

Table of Figures

1. Introduction

Virtualization, per say, is an act of creating a virtual version of something including but not limited to a virtual hardware platform, operating system, storage devices or network resources[1]. It is an old concept which is being utilized with greater capacities as the demand for high performance and high throughput computing has increased rapidly in the recent years.

A model utilizing the immense potential of virtualization and the unprecedented success of internet is 'Cloud Computing'. "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction" [2]. It has a service-based architecture and provides services on an infrastructure, platform and software level. Even though cloud computing has been widely accepted commercially as well as a research prospect and continues to grow as a vital model in IT, there remains scope of improvement due to challenges faced in the domain of virtualization, scalability, interoperability, quality of service, failure handling etc.

The performance overhead of virtualization has decreased substantially owing to improvisation in hardware. Hence, there has been an increase in the demand for virtualised systems that can handle larger workloads, both CPU as well as I/O intensive. I/O is a dominant factor in assessing computing performance; hence I/O virtualization becomes an in extractible feature of virtualization framework. For high I/O rates, the CPU overhead for handling all the interrupts can get very high and eventually lead to lack of CPU resources for the application itself [3]. CPU overhead becomes more evident when we are working in a virtualized scenario, trying to accommodate as many virtual machines as possible in one physical machine.

In the cloud environment, a virtualized system consists of a hypervisor that manages initiation, execution and management of the virtual machines on a physical host machine. It is responsible for the multiplexing and scheduling of shared resources between the hosted virtual machines. A few examples of the existing hypervisors include VMware ESX server, Xen and KVM. In this particular project, focus has been maintained on KVM hypervisor. KVM is a Linux kernel-integrated hypervisor which makes use of scheduling and memory management capabilities of Linux kernel. It utilizes the functionalities of QEMU for emulating I/O requests of the guest OSes and Virtio drivers for communication with physical devices.
The emulation and execution of interrupts to/from guests is accomplished with the collaboration of QEMU, KVM and Virtio.

To narrow down further, interrupts generated due to network have been considered. As the network bandwidth increases, interrupt due to packet readiness also increase. For each packet to be sent from or received by guest, an interrupt is triggered which traps in hypervisor.

Another important feature of this research is the focus on Intel x86 architecture as it is one of the most widely used architecture in current scenario. It also provides the benefit of hardware assisted virtualization which reduces CPU overhead by a great extent. This thesis, hence, focuses on the need to improve handling of virtual network I/O interrupts to make virtualized systems faster. The first few sections elaborate on the concerned terms and software libraries that were a part of the literature survey. The next section is concerned with the techniques that have been used to handle interrupts efficiently. Further, the designed algorithm has been stated with time and space complexity analysis. This is followed by case studies of interrupt handling in real time systems with different features. The report ends with the contribution of the student to existing technology and the future scope of the consummated research.

2. Cloud Computing

A cloud is a pool of virtualized computer resources. It is both network centric and data intensive. An Internet cloud of resources is a distributed computing system which applies parallel and distributed computing. They can be built with physical or virtualized resources over large data centres that are centralized or distributed [4]. Fig.1 gives an idea about the evolution history of the concept of cloud computing. A cloud can host a variety of workloads including batch style backend jobs and interactive user applications. Cloud allows workloads to be deployed and scaled out quickly through rapid provisioning of virtual/physical resources. Cloud computing applies a virtualized platform with elastic resources on demand by provisioning hardware, software and data sets dynamically [4]. It basically merges two established concepts, statistical multiplexing and virtualization of resources, to provide computing as a utility.

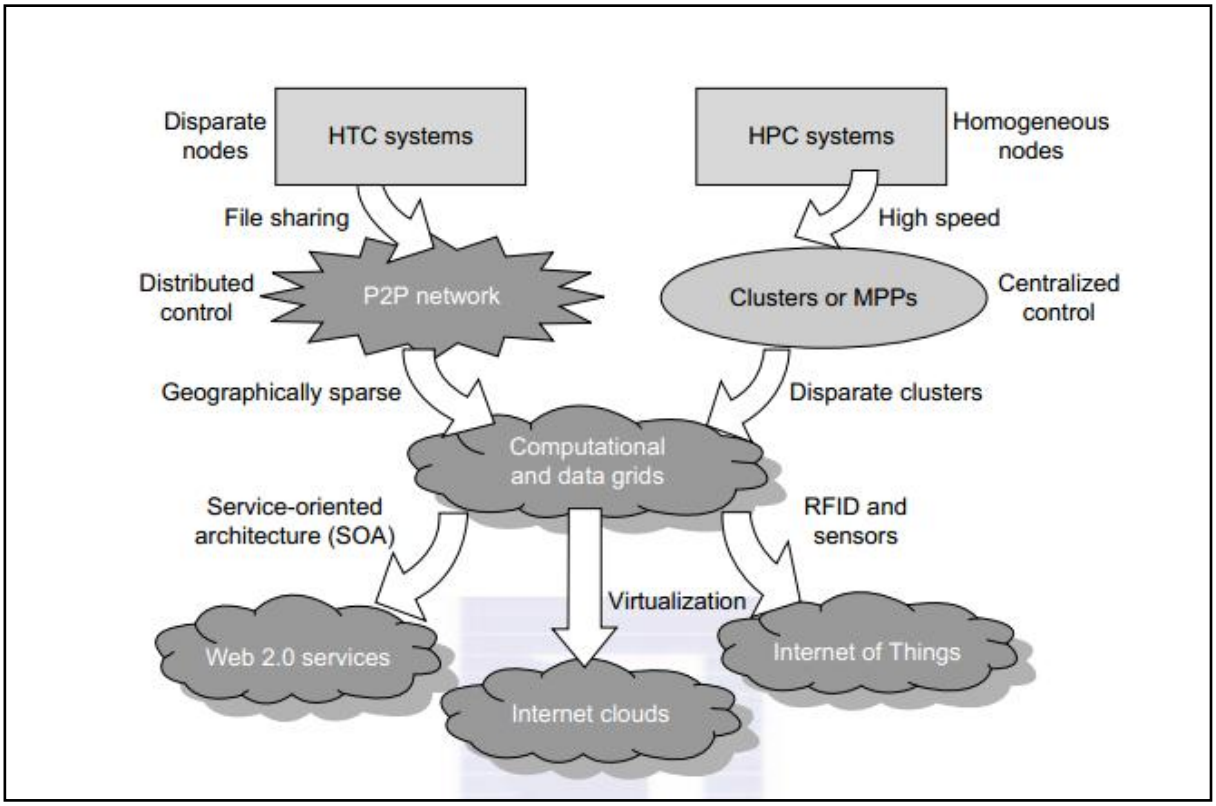


Figure 1 Evolutionary trend toward cloud computing, Internet of Things and Web 2.0 Services including Clusters, Grids, MPPs [4]

Cloud provides three basic kinds of services:

1. IaaS - It puts together infrastructure namely, servers, storage, networks and data centre requirements. Users can deploy VMs running guest operating systems and run specific applications. E.g. Amazon EC2.
2. PaaS - It provides a virtualized platform to users to deploy user built applications. It puts together middleware, database, development tools and runtime

support. The provider of the cloud supplies API and software tools. E.g. Google AppEngine.

3. SaaS - It puts together infrastructure, development environment and browser initiated application software. Users can run their data across these applications. E.g. Salesforce.

Cloud computing supports redundant, self recovering, highly scalable programming models. It maximizes resource utilization through massive parallelism, job throughput and provides storage and power efficiency. It deploys real time resource monitoring to enable rebalancing of allocated resources when ever needed, hence leading to energy conservation. Typical cloud architecture is shown in Fig 2.

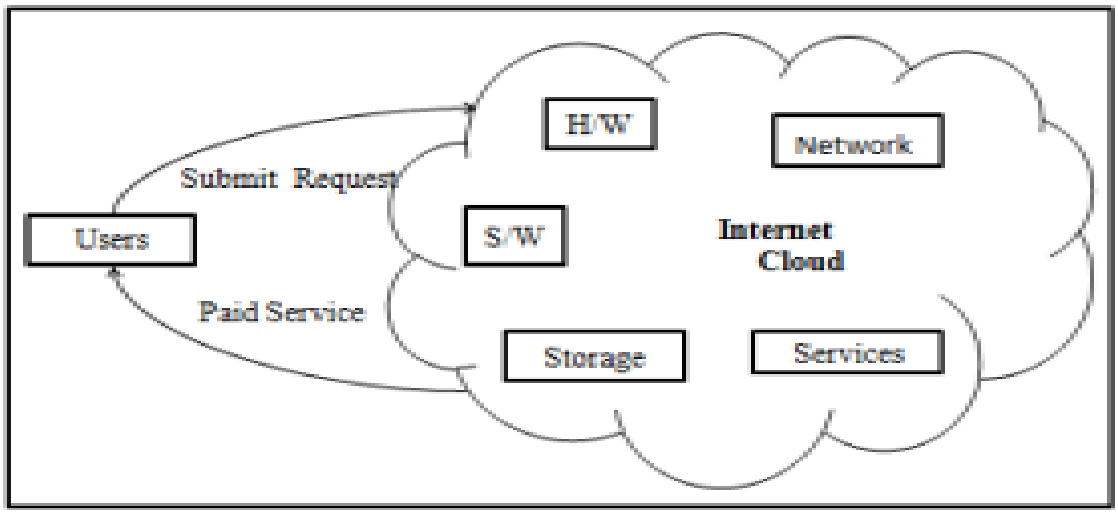


Figure 2 Basic Cloud Architecture [4]

Cloud computing provides transparency in data access, resource allocation, process location, concurrency in execution, job replication and failure recovery, both to users and system administrator [4]. It is ubiquitous in the sense that it is reliable and scalable, autonomic such that it is dynamic and discoverable and compassable such that it provides services in accordance with Service Level Agreement with best possible Quality of Service.

A cloud, hence, provides three basic functionalities [18]:

1. Illusion of infinite resources available on demand. It eliminates the need for cloud computing users to plan ahead for provisioning.
2. Elimination of an upfront commitment by cloud users. It allows users to start small and increase their hardware resources as per increase in needs.
3. Ability to pay for use of computing resources on a short term basis as needed and release them as needed.

3. Virtualization

A virtual machine is a software abstraction that behaves as a complete hardware computer, including virtualized CPUs, RAM and I/O devices [5]. Virtualization, in our context, is a computer architecture technology by which multiple virtual machines are multiplexed in the same hardware machine ensuing a logical division of available physical resources. It allows multiple guest operating systems to manage their numerous applications on the same hardware without any support from the host operating system. Fig. 3 below gives a basic view of a virtualized system hosting two virtual machines with their own applications, operating systems and emulated hardware devices. A software layer resides between the virtual machines and the host OS/hardware (in case of a bare-metal approach) known as Hypervisor.

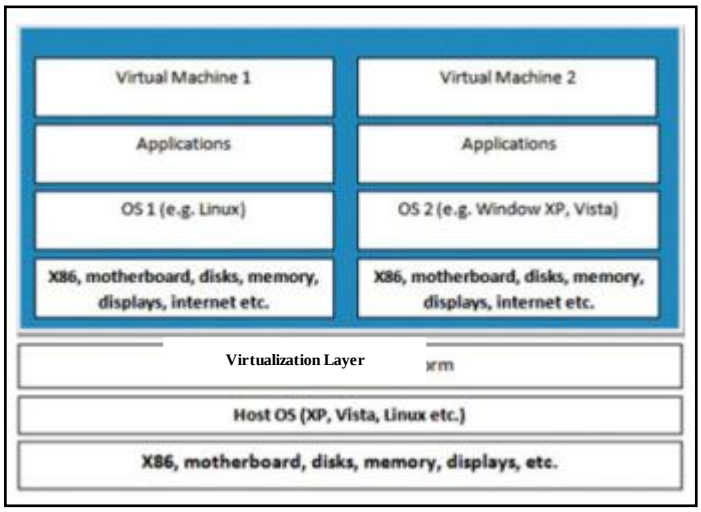


Figure 3 Virtualized system with two hosted virtual machines [19]

After virtualization, different user applications managed by their own guest operating system run on same hardware independent of the host operating system. This is often done by adding additional software called Hypervisor or Virtual Machine Manager. The main function of a hypervisor is to virtualized physical hardware of a host machine into virtual resources to be used by the virtual machines exclusively. The term guest is commonly used to distinguish the layer of software running within a VM; a guest operating system manages applications and virtual hardware, while a hypervisor manages VMs and physical host hardware [5]. It handles the deployment, management and functioning of VMs and is responsible for allocation of hardware resources to VMs. In a virtualised system, it is not possible for a virtual machine to access any resource that has not been explicitly allocated to it. Some of the widely used hypervisors include Xen and KVM.

There are three kinds of virtualization technologies, namely, full virtualization, para-virtualization and hardware assisted virtualization. These have been discussed in detail in the following subsections.

In a traditional system with only one host OS, the execution of instructions is as shown in Fig. 4. User level instructions execute in user mode while certain privileged instructions involving changes to memory management structures, system calls et al trigger a context switch to kernel mode and then execute.

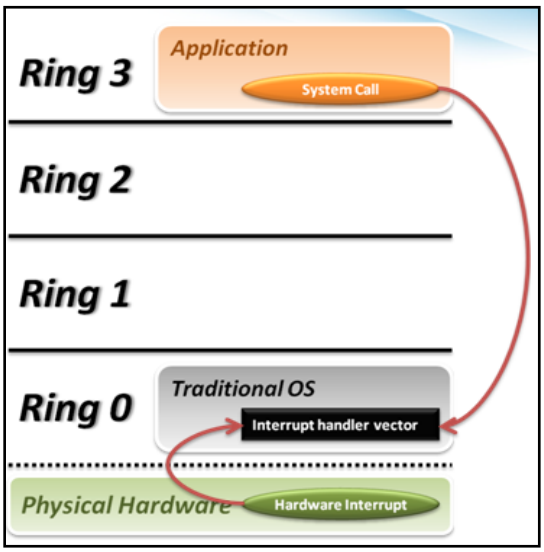


Figure 4 Instruction execution in a traditional system with one host OS [22]

Owing to the traditional architecture of Intel x86, rings 1&2 have been shown in Fig. 4. However, in modern systems, there are only two privilege levels, user mode and kernel mode. Intel uses the rings 1 and 2 for device drivers, guest operating systems (in case of virtualization) and I/O privileged code [20] so that they remain privileged but are separated from the rest of the kernel.

3.1 Techniques of Virtualization

3.1.1 Full Virtualization

It involves complete emulation of underlying hardware to allow the guest operating system to run unmodified. It depends on trapping and virtualization of certain non-virtualizable resources and privileged instructions. Full virtualization traps the privileged instructions of the guest OS and utilizes binary translation to virtualize their execution. However, there are certain instructions in Intel x86 like popf/push which are not trapped by the CPU. In this case, the hypervisor browses the memory of the virtual machine and rewrites these instructions before they are executed. Other non-privileged, user level instructions continue to execute unmodified. Fig. 5 below gives a brief look into the system architecture for a fully virtualized machine.

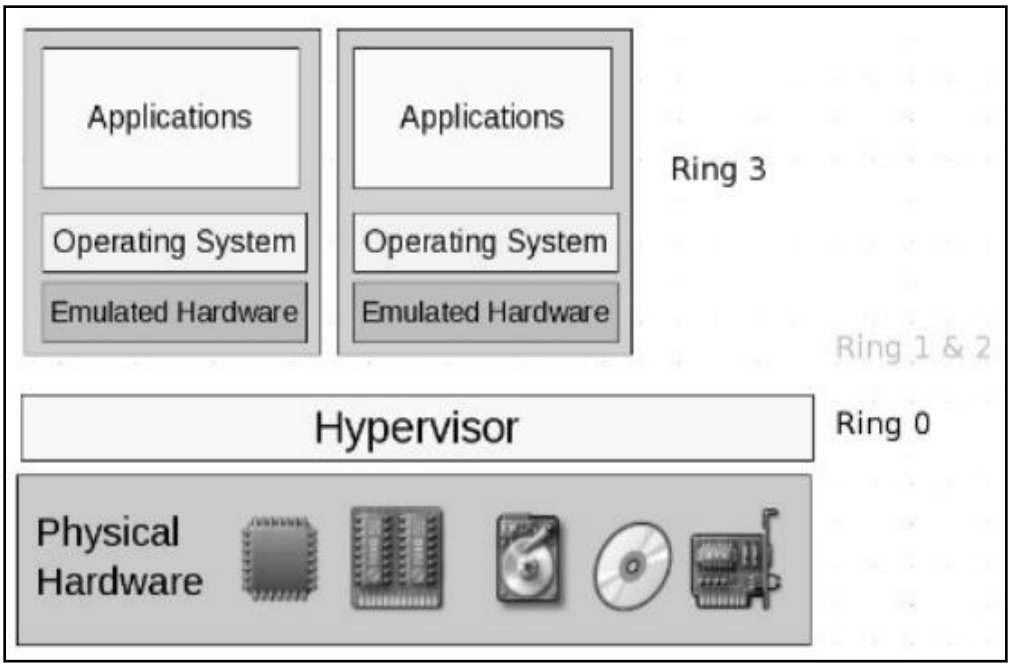


Figure 5 Full virtualization [21]

3.1.2 Para Virtualization

The underlying hardware of the system is not emulated. Guest operating systems are allowed to run in their isolated domains giving a sense of exclusivity. However, guest kernel is modified to replace non-virtualizable instructions with a call to the hypervisor, which further handles these instructions. The Para-virtualized guest OS uses an intelligent compiler to convert sensitive instructions to hypercalls. Fig. 6 below shows the architecture of a Para-virtualized system.

The figure includes the depiction using Xen hypervisor, which is currently the most extensive example of Para-virtualized hypervisor models. Xen is a bare metal hypervisor which does not need a host OS to execute. In this system, there is a privileged guest OS which has direct control over the physical hardware of the system. Rest of the virtual machines are managed and allocated resources through intervention by the privileged VM. This technique of virtualization is faster than full virtualization as it escapes the burden of binary translation of instructions.

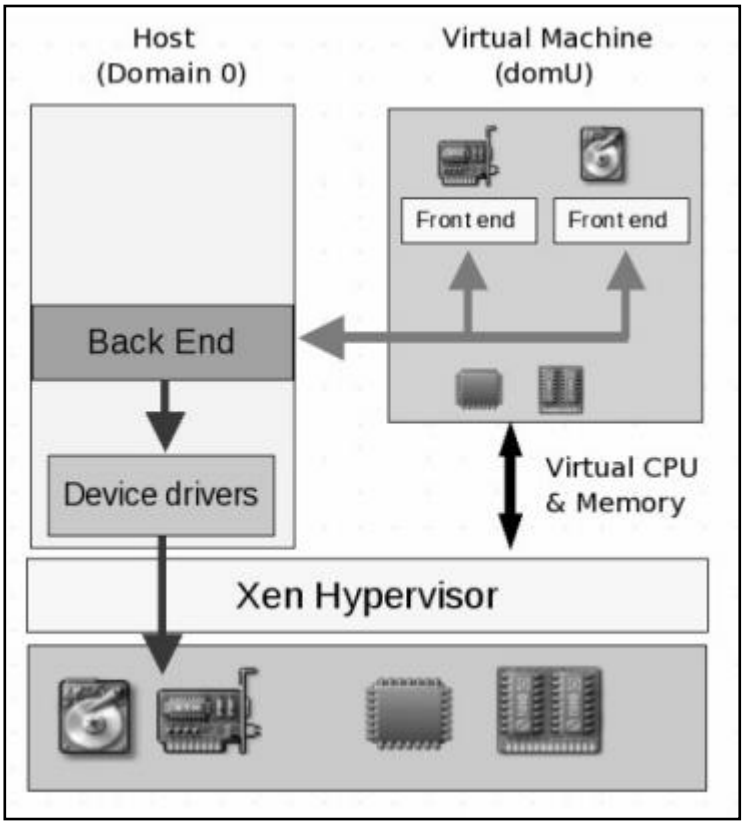


Figure 6 Para Virtualized System with Xen Hypervisor [21]

3.1.3 Hardware Assisted Virtualization

This technology employs specially designed CPUs and other hardware components to improve the performance of hardware virtualization. The implementations of these features by Intel (VT-X) include providing a new operating mode to the CPU which can now operate in host mode or guest mode. A hypervisor can request that a process operates in guest mode, in which it will still see the four traditional ring/privilege levels, but the CPU is instructed to trap privileged instructions and then return control to the hypervisor [21]. Using these new hardware features, a hypervisor does not need to implement the binary translation that was previously required to virtualize privileged instructions. Direct memory access support is also provided to the guest by Intel so that the guest can access its assigned memory without hypervisor intervention. Translation buffers are maintained in hardware keeping the information of guest virtual to physical machine memory mapping.

3.2 Different Faces of Virtualization in Cloud Environment

Virtualization in a cloud environment is accomplished in a number of ways, including CPU virtualization, memory virtualization and I/O virtualization which are explored in detail in the next few sections of the report.

3.2.1 CPU Virtualization

CPU virtualization corresponds to over-commitment of CPU resources. A single CPU core is shared between different virtual machines with the help of scheduling algorithms as applied by the hypervisor. When instructions of a virtual machine are trapped by a hypervisor, it acts as a unified mediator for hardware access from different virtual machines to ensure correctness and stability [4]. Fig. 7 below shows the basic architecture of CPU virtualization with Intel VT-x support.

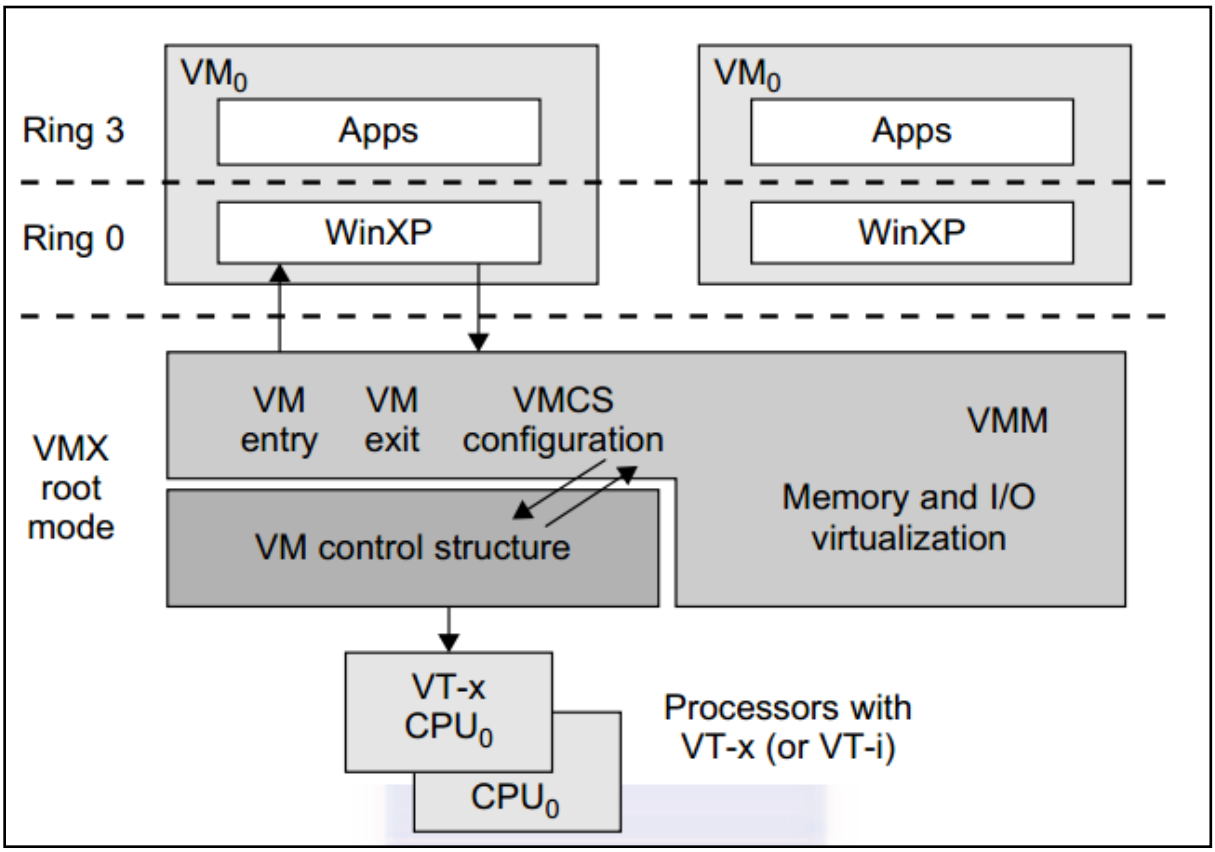


Figure 7 CPU virtualization with Intel VT-x hardware virtualization support [4]

3.2.1.1 Software based CPU Virtualization

In this scenario, the guest application code runs on the processor while the guest privileged code is translated and the translated code executes on the processor [23]. Privileged instructions include system calls, page table updates etc.

3.2.1.2 Hardware assisted CPU Virtualization

In this scenario, the guest uses guest and privileged modes of execution. The guest application code runs in guest mode. However, in case of sensitive instructions, processor undergoes a context switch from guest mode to root mode. The hypervisor executes in the root mode and executes appropriately to handle the trapped instructions without translating the code. The guest is then restarted in the guest mode.

3.2.2 Memory Virtualization

All modern day operating systems provide virtual memory support by utilizing the concept of paging, thus, allowing applications to collectively use more memory than is physically available. In a similar way, hypervisors provide virtualization support for over commitment of machine memory. The guest memory configured for all virtual machines might be larger than the amount of physical host memory [23]. Memory virtualization is depicted in Fig. 8.

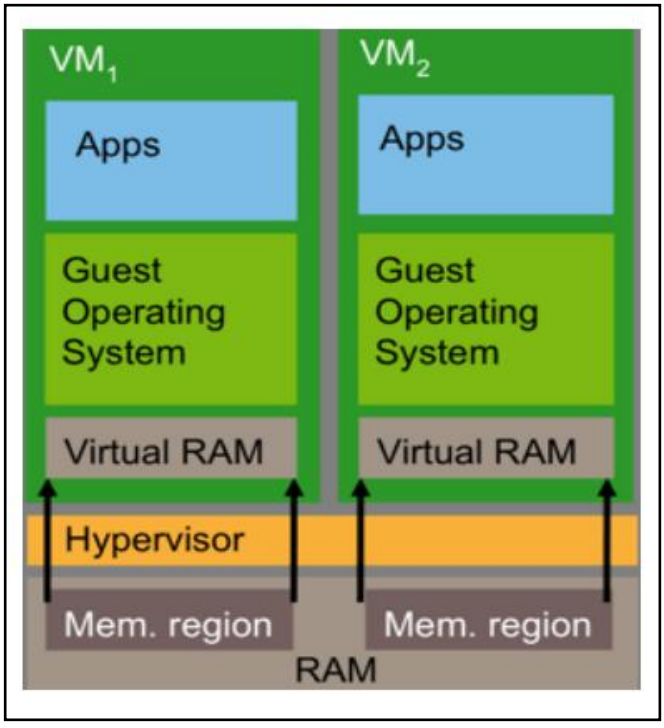


Figure 8 Memory management in Virtual Machine [25]

In memory virtualization, physical system memory is dynamically allocated to the virtual machines as guest physical memory. The hypervisor dedicates part of the system memory for its own use while some memory is required for the code and data of virtual machines.

Moreover, each virtual memory also requires its own memory for its application. Virtualization increases guest physical memory by adding an extra level of address translation. Hence there exists a two stage mapping as shown in Fig. 9. One from Guest Virtual Memory to Guest Physical Memory, which is taken care of by the guest OS. Second from Guest Physical Memory to Host Physical Memory, which is taken care of by the hypervisor.

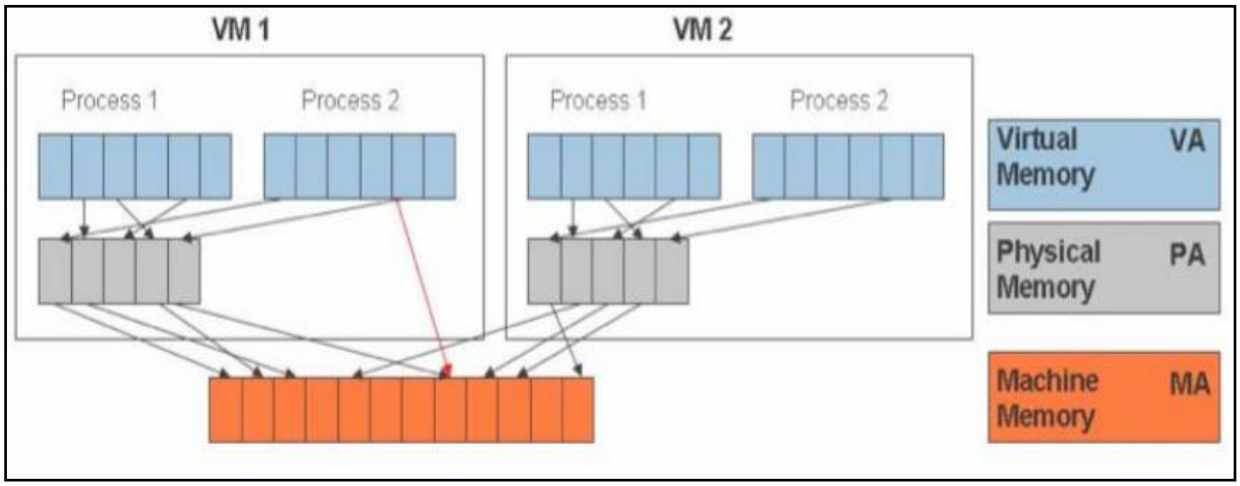


Figure 9 Two stage mapping for Memory Virtualization [4]

3.2.2.1 Software based Memory Virtualization

Hypervisor for each VM maintains a mapping from guest OS's physical memory pages to machine memory pages on underlying machine. VM sees a 0 based addressable physical memory space. Hypervisor intercepts VM instructions that manipulate guest OS memory management structure so that actual MMU on processor is not updated by VM. It maintains a shadow of the VM's page table. The shadow page table controls which pages of machine memory are assigned to a given VM. When OS updates its page table, hypervisor updates the shadow.

3.2.2.2 Hardware Assisted Memory Virtualization

Processors use TLB hardware to map the virtual memory directly to the machine memory to avoid the two levels of translation on every access. When the guest OS changes the virtual memory to a physical memory mapping, the hypervisor updates the shadow page tables to enable a direct lookup [26].

3.2.3 I/O Virtualization

I/O virtualization is an essential part of the virtualization framework. It is a dominating factor in measuring computing performance and I/O bottlenecks affect both performance and throughput, aggrandizing the effect if the application is I/O intensive. In a system, having the ability to run multiple virtual machines, I/O virtualization manages the routing of I/O requests between virtual devices assigned to virtual machines and shared physical devices. There are three prevalent ways to accomplish I/O virtualization:

1. Full device emulation

2. Para virtualization
3. Direct I/O virtualization

3.2.3.1 Full Device Emulation

This technology involves emulation of well known real world devices. All functions of a device/ bus infrastructure like enumerations, identifications, interrupts, direct memory access are replicated in software.

Full device emulation architecture consists of:

1. Guest device driver
2. Virtual device
3. Communication mechanism between virtual device and virtualization stack.
4. Virtualization I/O stack
5. Physical device driver
6. Real device

Functions of virtualization I/O stack include:

1. Translation of guest I/O address to host address.
2. Inter VM communication.
3. Multiplexing I/O requests from/to physical devices.
4. Provide I/O features like cow disk etc.

The real device can be different from the emulated virtual device. Fig. 10 gives the basic architecture of full device emulation I/O virtualization.

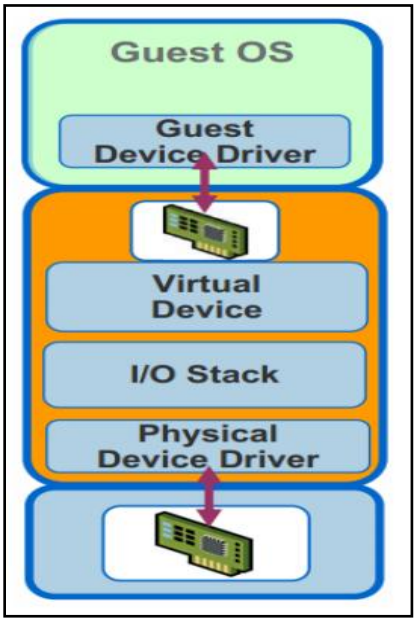


Figure 10 Full I/O Virtualization[6]

3.2.3.2 Para I/O virtualization

The most well-known example of Para-virtualization of I/O devices is Xen. This technology makes use of two device drivers, one employed in the front end or the user deployed virtual machine and the second in the backend or the privileged virtual machine. The front end device

Driver manages the I/O in the guest operating system. The backend device driver is responsible for managing real I/O device and multiplexing the I/O data of different virtual machines.

Various signalling or messaging mechanisms for inter VM communication used by different vendors for Para-virtualization include:

1. VMware : virtual PCI bus and IPC or syscall
2. Xen/Microsoft: XenBus/VMBus,IPC

This technique however, incurs greater CPU overhead than full device emulation. Fig. 11 gives the basic architecture of Para I/O virtualization.

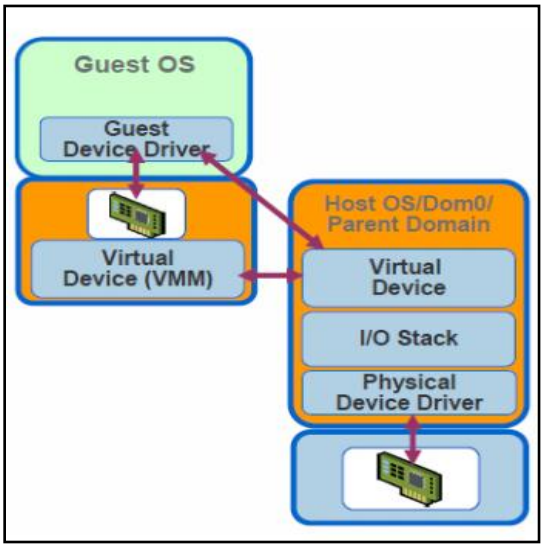


Figure 11 Para I/O Virtualization [6]

3.2.3.3 Direct I/O virtualization

This technology allows the virtual machines to access I/O devices directly. I/O devices are assigned to spaces in memory called domains to which isolated DMA access by virtual machine is ensured with the help of DMA remapping hardware provided by Intel VT-d. Rather than invoking the hypervisor for all (or most) I/O requests from a partition, the hypervisor is invoked only when guest software accesses protected resources (such as I/O configuration accesses, interrupt management, etc.) that impact system functionality and isolation [24]. Fig. 12 gives the basic architecture of direct I/O virtualization.

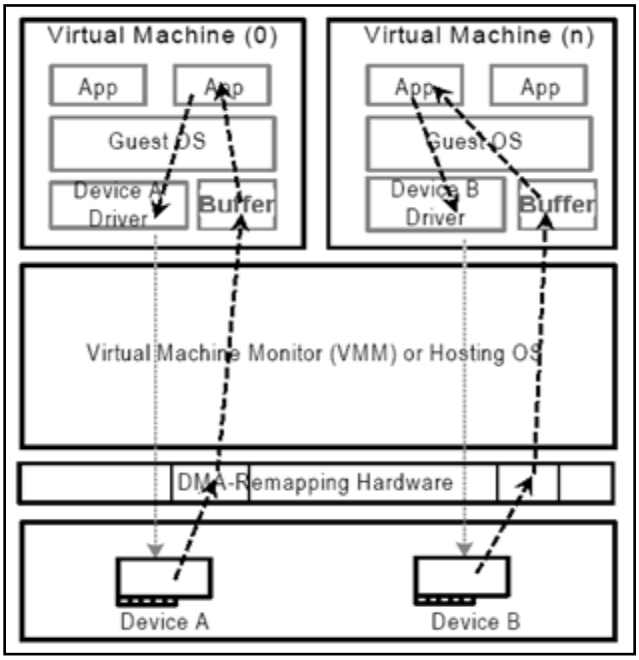


Figure 12 Direct Assignment of I/O devices [24]

4. Interrupts, Device Drivers and Interrupt Handlers

4.1 Interrupts

An interrupt is a signal that a hardware or a software emits to the processor in order to indicate an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing, the current thread. The processor responds by suspending its current activities, saving its state, and executing a small program called an interrupt handler (or interrupt service routine, ISR) to deal with the event. This interruption is temporary, and after the interrupt handler finishes, the processor resumes execution of the previous thread [7]. There are three kinds of interrupts:

Hardware Interrupts - These are generated by hardware devices associated with the system, either contained within or external peripheral. These are asynchronous and can occur in the midst of an instruction, hence requiring additional care in programming.

Software Interrupts - A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed [7]. These are akin to subroutine calls and help in making requests for services from low level software such as device drivers. For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to the disk [7].

Whenever an interrupt is encountered, processor does a context switch, handles the current interrupt with a program called interrupt handler and then returns back to its saved state i.e. the punctuated instruction. Each interrupt has its own interrupt handler. Each operating system family has its own interrupt descriptor table I/O devices generate asynchronous interrupts to communicate with the processor completion of I/O operations. Its main advantage is that the processor can continue to execute other instructions until it is interrupted instead of polling each hardware device individually in a timely manner. However, in a virtualized system deploying multiple virtual machines, each interrupt results in a costly exit from the guest operating system to the host system, regardless of whether the device is assigned or not.

The flow of I/O request in a virtualized system is as follows (see Fig. 13 for a diagrammatic representation) [27]:

1. Application running within virtual machine issues an I/O request.
2. The I/O stacks in guest OS processes this request.
3. A device driver in the guest OS issues this request to a virtual device. This is a privileged instruction and hence, gets trapped by the hypervisor.
4. Hypervisor schedules the requests from multiple virtual machines onto an underlying shared physical I/O device, usually via another device driver maintained by the hypervisor.
5. When physical device finishes processing the I/O request, the two stacks must be traversed

again but in reverse order.
6. The actual device posts a physical completion interrupt which is handled by the hypervisor.
7. Hypervisor determines which virtual machine is associated with the completion and notifies it by posting a virtual interrupt for the virtual device managed by the guest OS.

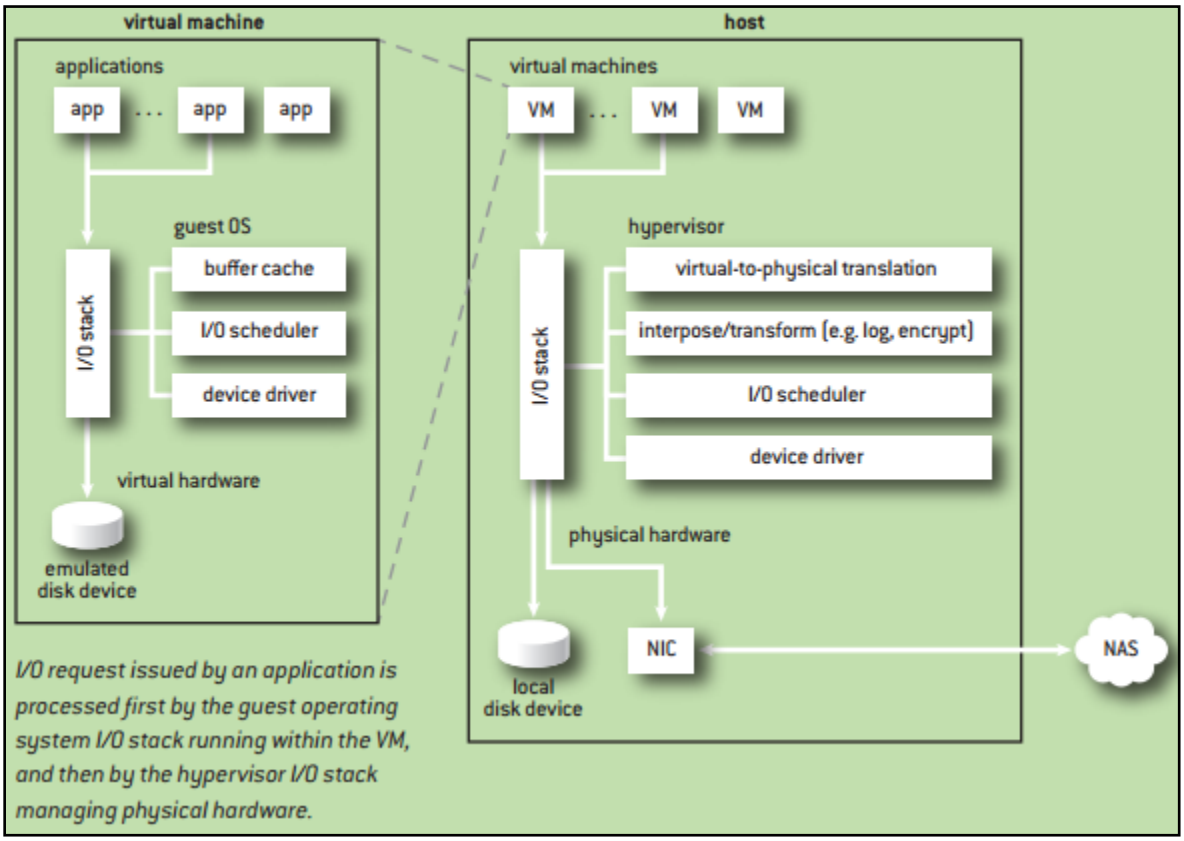


Figure 13 Flow of an I/O request in a virtualized environment [27]

The context switches between the guest and host caused by interrupts induce a considerable CPU overhead, thus adversely affecting guests that require throughput of as little as 50 Mbps [3]. The Turtles project [9] shows interrupt handling to cause a 25% increase in CPU utilization for a single-level virtual machine when compared with bare metal, and a 300% increase in CPU utilization for a nested virtual machine. Hence, accelerating guest code by improving device virtualization, specifically, virtual interrupt delivery, will help provide a better throughput and performance in a cloud environment.

In x86 architecture, every interrupt is assigned a code to handle itself. This code pushes the interrupt number on the PIC stack and jumps to a common segment called 'do_IRQ'. 'do_IRQ' acknowledges the interrupt and obtains a spinlock for the given number, thus, preventing any other CPU from handling this IRQ. It also clears up certain status bits (IRQ_WAITING) and looks up the handler for this IRQ [30].

In this report, our focus will remain on interrupts generated due to network traffic. Each time the guest OS is supposed to receive/transmit a network packet, an interrupt is generated. As the network bandwidth increases, the frequency of interrupts due to network also goes up. For

instance, a 10 Gbps network can receive up to 0.8 million 1.5 Kb packets per second, and it may have a 10X higher frequency for smaller packets [28].

4.2 Existing Techniques for Efficient Interrupt Handling

Interrupt handling has been a topic of research for many years now. Various technologies to reduce processing overhead due to context switches applicable to both bare metal as well as virtualized systems have been developed. This section talks about some of the existing techniques for the same.

One of the earliest innovations to reduce processing overhead was a technique known as Polling [11]. As the name suggests, polling completely overrides the interrupts and the device is periodically polled for any new events. This is beneficial in the sense of synchronous management of device events thus letting the OS to make decisions regarding where to poll and hence keep a check on the number of handler invocations. The limitations are the time delay and redundant cycles when no events are pending. Although latency improves when polling is done on a different core, a core gets wasted. Power consumption is also more since the processor never becomes idle.

Another method suggested to reduce interrupt handling overhead is hybrid approach [17]. Hybrid approach dynamically switches between polling and interrupt delivery. Linux by default uses this approach of interrupt handling. However, this approach does not work well always, as it is difficult to predict the number of interrupts a device will issue in a given time.

Interrupt coalescing [12] approach to mitigate the effect of numerous context switches is to program the device to send one interrupt in a time interval or one interrupt per several events, as opposed to one interrupt per event [3]. However, to model the coalescing algorithm and to accurately predict the parameters is crucial as wrongful implementation might lead to an increase in latency.

One of the recent approaches to handle interrupts efficiently is ELI or Exit Less Interrupts [3]. As opposed to coalescing which reduces number of interrupts, ELI focuses on reducing the number of context switches between guest and host. ELI delivers the interrupts directly to guests for the devices assigned to them without the host intervention. All other interrupts are transferred to the host. It accomplishes this by maintaining a shadow page table for each of the interrupts triggered by the devices directly assigned to it. This technique, however, is still undergoing research.

4.3 Device Drivers

A device driver is a computer program that operates/controls a particular type of device that is attached to a computer. A driver typically communicates with the device through the computer bus or communication subsystem to which the hardware connects [29]. The host OS or hypervisor invokes the interrupt handler routine in device driver which further invokes the device to perform the appropriate action.

In a virtualized environment, virtual device drivers are used. These device drivers prevent communication between guest OS and the underlying hardware by emulating a piece of hardware. It is responsible for creating the impression that guest OS and its drivers, running in a virtual machine, have access to the physical hardware. Attempts by the guest OS to access the hardware are routed to the virtual device driver in the host OS [29]. Virtual device drivers are also responsible for injecting virtual interrupts into guest OSes running on the various virtual machines.

4.3.1 Physical Network Driver

Network driver in an operating system is responsible for handling interrupts generated due to network traffic or faults in receiving or transmission of packets by the physical NIC. Network interface must register itself within the specific kernel data structures in order to be invoked when packets are exchanged with the outside world [30]. However, network interfaces don't have an entry point like disk drivers neither the normal file operations like read and write make any sense for them. Since same physical network interface can host several sockets, network devices, once probed, seek kernel permission to push incoming packets towards the kernel. The main functions of a network device driver include setting addresses, modifying transmission and receiving parameters, maintaining traffic and error statistics.

In Linux kernel, device drivers need to poll for pending interrupts in fixed intervals of time. In Intel x86-64 for Linux kernel, device driver codes reside in arch/i386/kernel in modules 'irq.c', 'apic.c', 'entry.s' and 'i8259.c'. It makes use of two functions, namely, can_request_irq() and request_irq() for handling interrupts. can_request_irq() checks whether the particular interrupt pin on PIC is available to take requests or if it is shared between several devices and is currently being held by a sharing device. If the interrupt cannot be registered yet, it is masked till a pin becomes available. request_irq() is used to book the available PIC pin and hence, notify the device driver (when it polls) about the pending interrupt. Reported interrupts can be found in /proc/interrupts.

4.4 Interrupt Handlers

Interrupt handler is a routine that the device driver registers with the programmable interrupt controller to invoke when an interrupt for one of its device instances is acknowledged. The roles of an interrupt handler include [30]:

1. Gives feedback to its device about interrupt reception.
2. Read or write data according to meaning of interrupt being serviced.

3. Clearing a bit on interface board as most hardware boards will not generate other interrupts until their interrupt pending bit has been cleared.

The interrupt handler is responsible for awakening processes which are waiting for the corresponding event to happen if the interrupt from the device signals the event. However, there are certain limitations to the functionalities of interrupt handling codes. They cannot initiate data transfer to or from user space, cannot invoke a function call that will induce sleep and cannot invoke scheduling processes.

In all the operating systems, interrupt handling processes are divided into two parts. Since, this project concerns itself with KVM hypervisor, the terminology used will be in correspondence with Linux kernel. The two parts of the interrupt handlers are known as the top half and the bottom half, as can be seen Fig. 14.

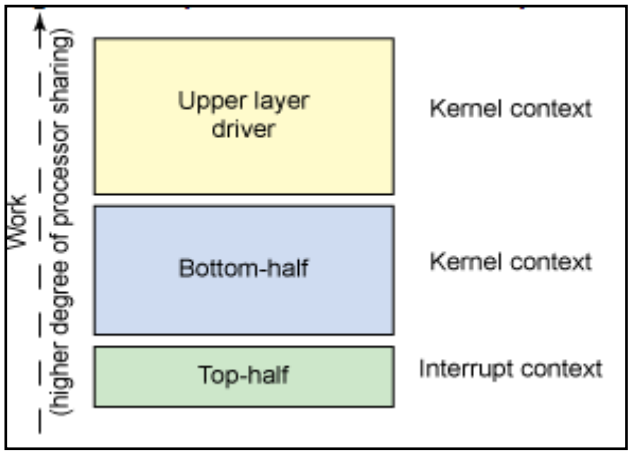


Figure 14 Top Half and Bottom Half Interrupt Processing in Linux Kernel [31]

The top half is the routine that is registered with request_irq() [30] and the one which actually responds to an interrupt. During this time, interrupts to this device are disabled. Once the top half exits, interrupts are again enabled. Bottom half is executed in the form of a tasklet, i.e., it is scheduled by the top half to be executed at some later point of time. Thus, the top half and bottom half execute concurrently and increase the speed of interrupt handling by making device available within a short interval of acknowledging the interrupt. In a typical scenario, the top half saves the device data to a device specific buffer, schedules the bottom half and exits [30]. The bottom half, when processed later, awakens the sleeping process and invokes another I/O if required.

For e.g., in case of a network interrupt, the network interface generates an interrupt. In case of receiving, the top half of the handler receives the data and pushes it into a buffer. The bottom half then processes the data when scheduled. The processor is interrupted by the hardware to signal arrival or transmission of data. The interrupt handler routine determines the scenario through a status register present on the physical NIC.

An interrupt handler's execution time consists of the following:

1. Hardware latency in generation and acknowledgement of interrupt, which is characteristic of a system.
2. Handler overhead in stacking and un-stacking the registers etc. which is fixed for a particular handler.
3. Execution time of the specific code of the handler e.g. tasks performed in the protocol stack of a network driver.
4. Operating system overheads to ensure correct pre-emption and context switching for execution of the interrupt handler code.

Interrupt latency is defined as the time interval from an interrupt event to the execution of handler code. Fig. 15 gives an overview of interrupt handling.

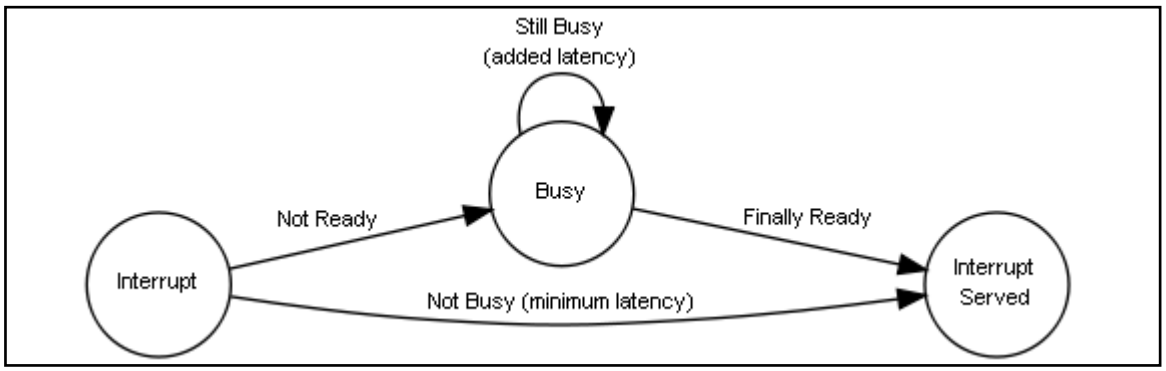


Figure 15 Interrupt handling procedure [33]

The jitter in the busy state corresponds to the sum of three factors [33]:

1. Higher or equal priority interrupt handlers execution time combined. This time can go from zero to the maximum randomly. This value can be guaranteed to be zero only if the interrupt has the highest priority in the system.
2. Highest execution time among lower priority handlers. This value is zero on those architectures (Cortex-M3 for example) where interrupt handlers can be pre-empted by higher priority sources.
3. Longest time in a kernel critical section that can delay interrupt servicing. This value is zero for fast interrupt sources which can pre-empt the kernel.

To optimize the latency due to the busy state, interrupt handler code is maintained to be as short as possible so that interrupts are not disabled for long. In this project, the technique of coalescing has been used to group interrupt completion messages of each virtual machine, hence, reducing the number of context switches between the host and guest.

5. I/O Virtualization in KVM

KVM [13,14,15] is a hardware assisted hypervisor integrated with the Linux kernel that enables a user to create and manage multiple virtual machines by multiplexing the physical resources. Its functionalities also include writing and reading from virtual CPU registers, injecting an interrupt into a virtual CPU and scheduling a virtual CPU. The basic ioctl()s that KVM uses to initiate the guest are: KVM_CREATE_VM, KVM_SET_USER_MEMORY_REGION, KVM_CREATE_IRQCHIP/../PIT, KVM_CREATE_VCPU, KVM_SET_REGS/../SREGS, KVM_SET_VCPU/..MSRS.../VCPU_EVENTS, KVM_SET_LAPIC and KVM_RUN. The basic architecture of KVM hypervisor is shown in Fig. 16.

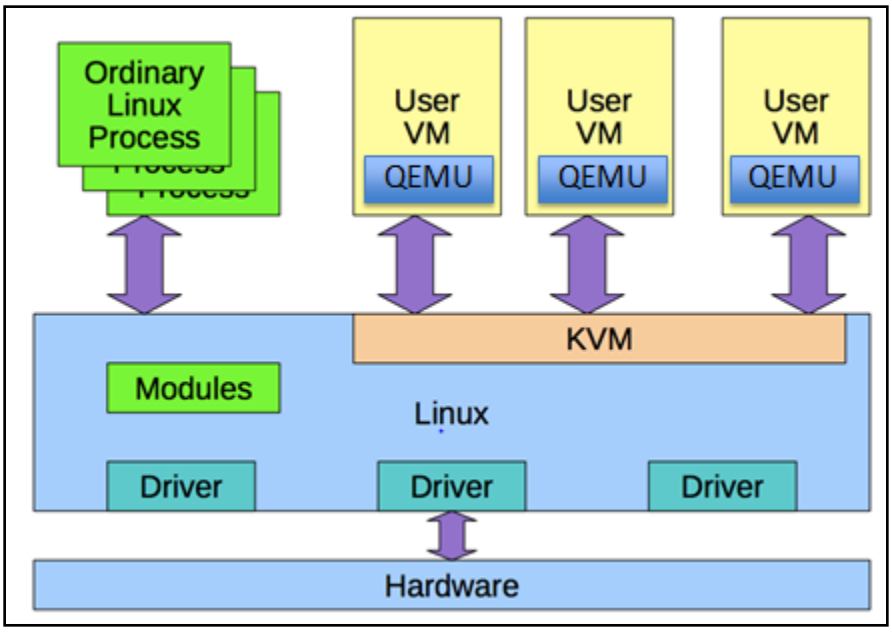


Figure 16 Basic KVM Architecture [40]

KVM uses QEMU and Virtio functionalities to perform virtualization of guest I/O requests. The guest operating system operates in guest mode. When it requests for an I/O, there is a context switch from guest mode to kernel mode as the guest exits to give control to KVM. The hypervisor determines the reason for the exit. If it is due to an I/O request, it switches to user space mode where QEMU executes to physically service and emulate the I/O request for the guest. Once serviced, it again signals the hypervisor to re-enter the guest mode with the help of an ioctl(). This procedure is shown in greater detail with a flowchart as is shown in Fg. 17.

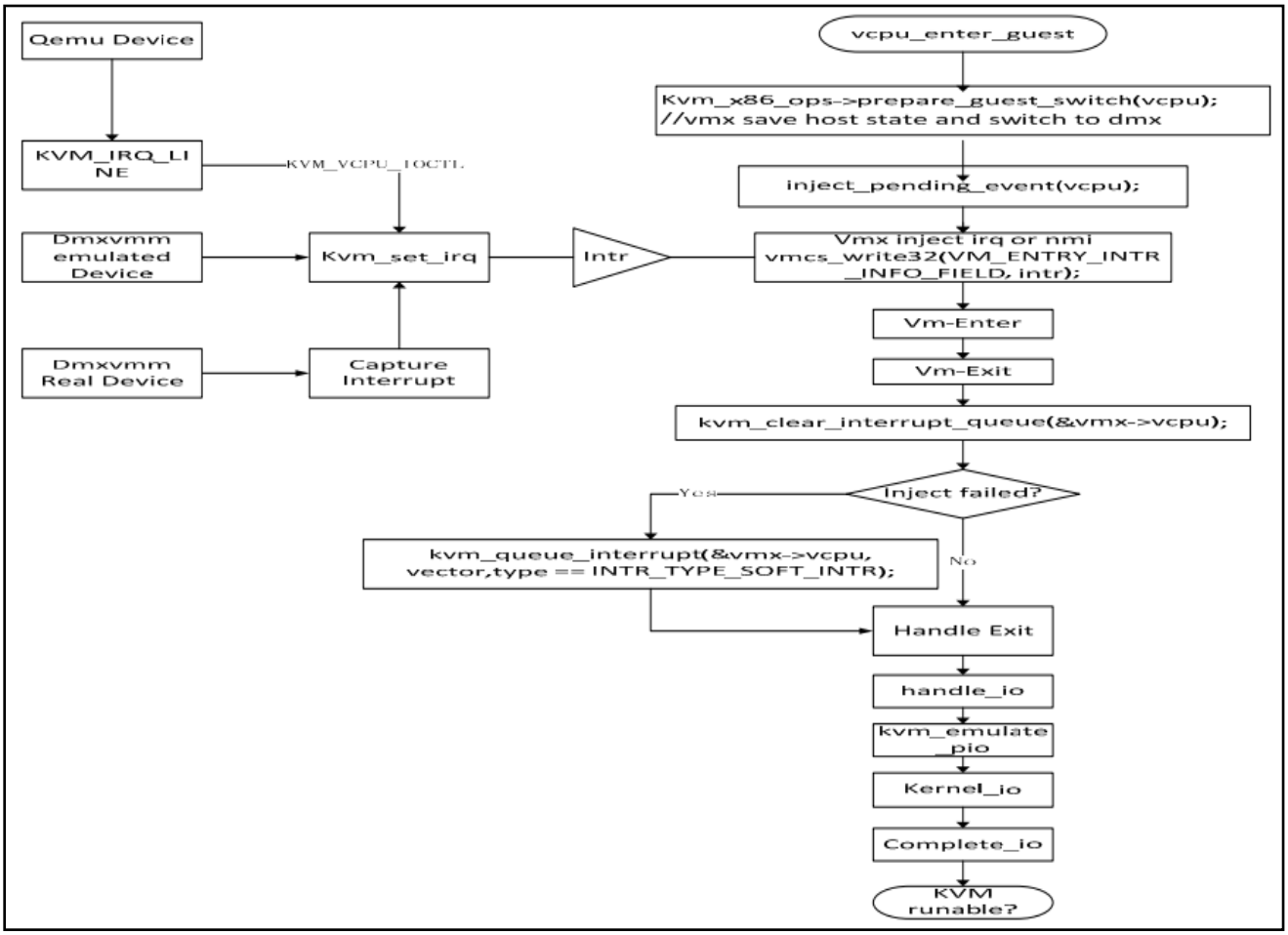


Figure 17 Interrupt delivery and Handling by KVM in a virtual machine [39]

As evident, the I/O stack of guest and host are traversed twice and there is CPU overhead of guest/host context switching. Since it is open source software, it will be ideal for carrying out the research process being aimed at. Integration with Linux will further help as a number of prefabricated instructions could be used.

The source code of KVM concerned with I/O virtualization, specifically interrupt injection and handling can be found in /linux/kernel/kvm/x86 in irq.c, lapic.c and vmx.c modules. KVM checks for timer events checks for non-APIC source interrupts, checks if vcpu has pending external interrupts, reads and acknowledges pending interrupts, injects virtual interrupts into vcpu, sets interrupt service register and interrupt request vector, sets the priority of interrupts in interrupt priority register. All of these functions identify the target vcpu through a VCPU_id which can be accessed through KVM_VCPU structure.

5.1 QEMU

QEMU (short for "Quick EMUlator") is a free and open-source hosted hypervisor that performs hardware virtualization. It provides a set of device models, enabling it to run a variety of unmodified guest operating systems [41] and uses KVM to take advantage of hardware assistance provided by Intel CPUs to run guest code directly on the real processors.

In association with KVM, QEMU resides in the user space and is responsible for device emulation and physically servicing the interrupt by interaction with the real devices. For each virtual machine that is launched, there is a dedicated QEMU thread which contains vcpu threads for each virtual CPUs assigned to the guest and a dedicated iothread that handles network packets and disk I/O operations. Whenever a guest application makes an I/O request, KVM exits to user space QEMU. QEMU then emulates the request and ensures that it is physically served. Basic architecture of QEMU can be seen in Fig. 18.

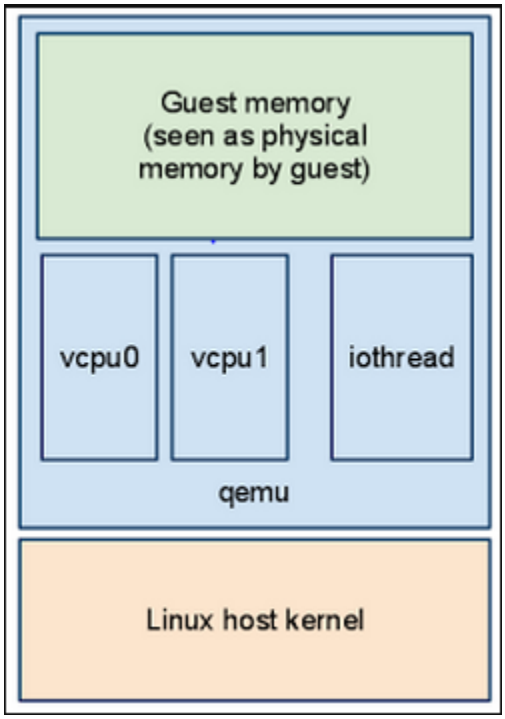


Figure 18 QEMU architecture [42]

QEMU inherently follows an event driven architecture but with KVM, it makes use of parallel architecture as well to make use of the multi cores available to it. The main event loop of QEMU is main_loop_wait() and it is responsible for three major processes [42]:

1. Waits for the file descriptors to become readable or writable. File descriptors play a critical role because files, sockets, pipes and various other resources are all file descriptors. File descriptors can be added using qemu_set_fd_handler().
2. Runs expired timers. Timers can be added using qemu_mod_timers().
3. Runs bottom halves, which are like timers that expire immediately. They are used to avoid re-entrancy and overflowing the call stack. These can be added using qemu_bh_schedule().

5.2 VIRTIO

Virtio was developed to provide generic device drivers for various hypervisors. Its goal was to provide a common Application Binary Interface for buffer publication and usage, viz. a viz. transport and to probe and configure devices. Virtio API relies on a simple buffer abstraction to encapsulate the command data needs of the guest [44].

In kernel virtio device emulation is provided by Vhost drivers for KVM. I/O access by guest is emulated by the user space process QEMU. One of the advantages of Vhost includes putting virtio emulation code into the kernel. This results into a direct kernel call by device emulation code instead of making context switches from user space. Whenever a virtual machine is initiated, a kernel thread called vhost-$pid is created by vhost driver, where $pid is the process pid of QEMU thread. This Vhost worker thread is responsible for handling I/O events and performing device emulation.

The vhost worker threads make use of file descriptors to signal I/O activity from or to the guest. Vhost instances have eventfd file descriptors which the worker threads monitor. The KVM kernel module has a feature known as ioeventfd for taking an eventfd and hooking it upto a particular guest I/O exit [44]. Similarly, when vhost worker thread needs to interrupt the guest, it writes to a "call" file descriptor. The KVM kernel module has a feature called irqfd which allows an eventfd to trigger guest interrupts. This process can be seen in Fig. 19 below.

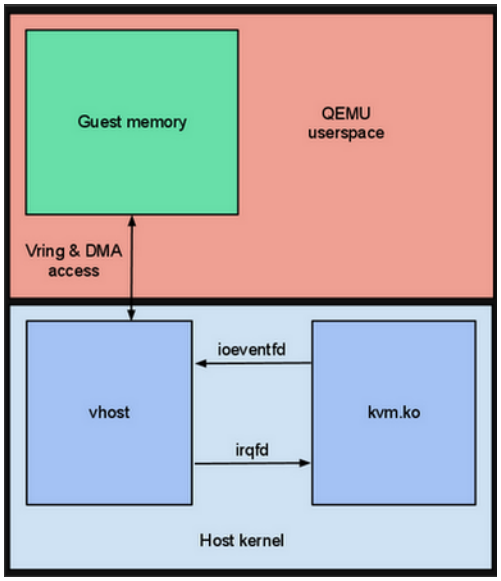


Figure 19 Vhost and KVM interaction for guest I/O events [45]

Virtio drivers on the guest side register themselves and specify the devices they can handle. Driver calls its probe function when a suitable virtio device is discovered. It then configures the device which is a four step process:

1. The device looks for specific feature bits corresponding to the features it wants to use [43] Feature bits are stated explicitly as the host has the knowledge of feature bits acknowledged by the guest.
2. Struct virtio_device is initialized which contains device specific information like mac address. Guest can both read and write to the structure.
3. Status of device probing is monitored through set and get operations.
4. A reset function to free the device is initialized. This particularly comes in use when driver module is temporarily loaded into the kernel.

The basic structure of Virtio stack is as shown in Fig. 20. Virtio uses two layers to facilitate communication between guest and hypervisor. The upper layer is the virtual queue interface through which back end and front end drivers are conceptually attached. Virtqueue is basically a queue into which host and guest post buffers for exchange. These buffers are technically in the form of scatter-gather lists consisting of readable and writeable parts [44]. These queues are actually implemented as rings.

As shown in figure, five front-end drivers are listed for block devices (such as disks), network devices, PCI emulation, a balloon driver (for dynamically managing guest memory usage), and a console driver. Each front-end driver has a corresponding back-end driver in the hypervisor [44].

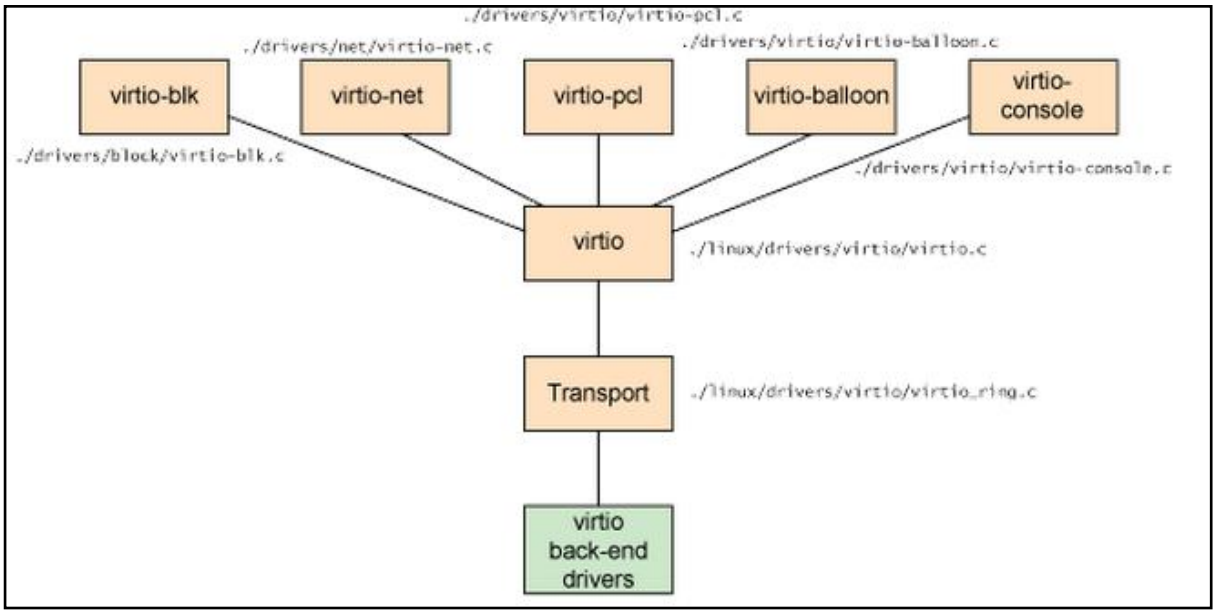


Figure 20 High level Architecture of VIRTIO [44]

The virtio front end as seen by the guest is shown in Fig. 21. The front device driver is represented by the Virtio_driver in the guest. The virtio_device is the emulated device pertaining to this driver. This is a virtual representation of the device in the guest. Each virtio_device has a corresponding Vitrio_config_ops module associated with them which defines operations for configuring these devices. Virt_queues are meant to serve these virtio devices by receiving and transmitting buffers. Virtqueue_ops are the main feature of this model. These form the foundation for communication between the guest and host device

driver. Each virtqueue corresponds to a virtqueue_ops structure that defines the basic operations for dealing with the hypervisor.

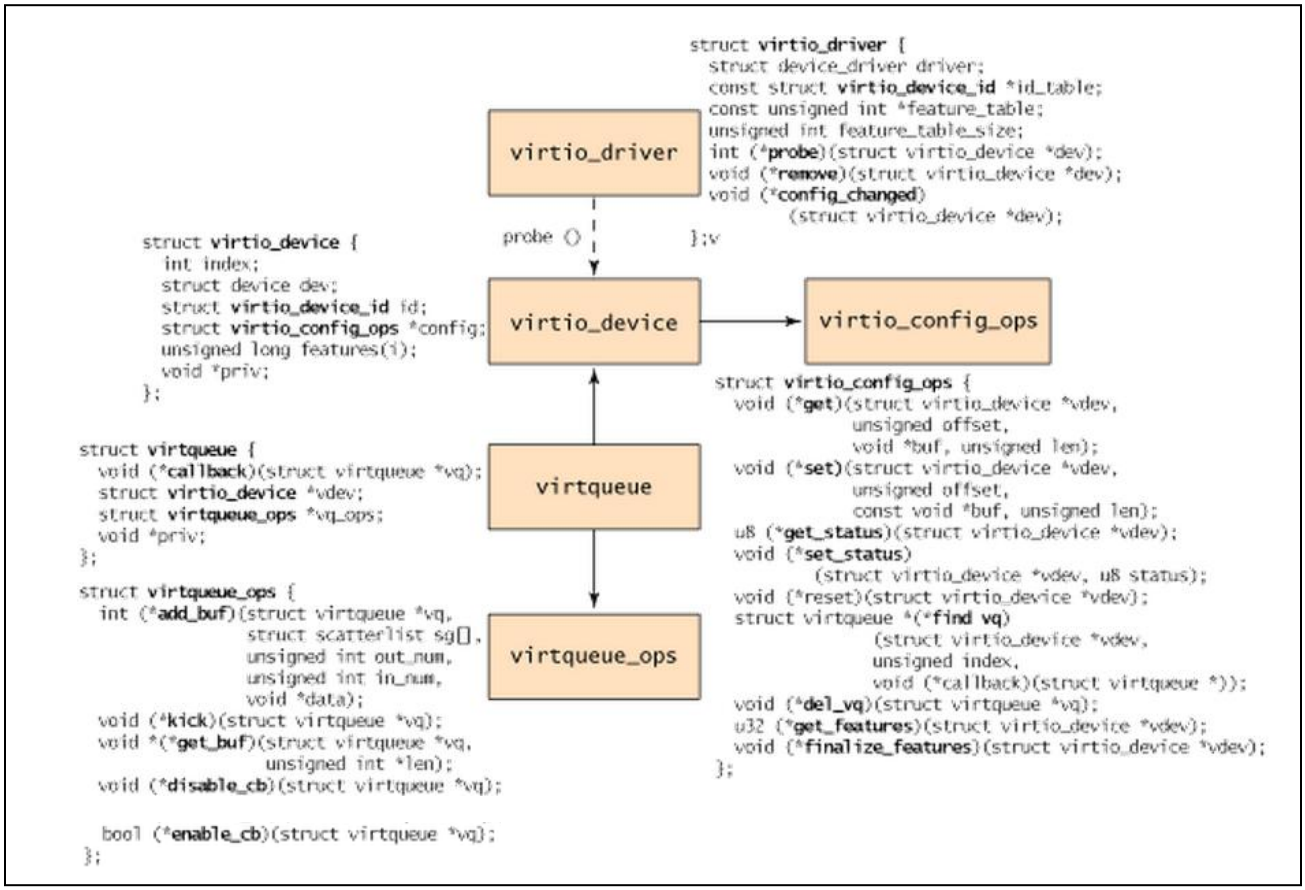


Figure 21 Virtio front end Architecture with corresponding data structures [44]

The virtqueue_ops function calls are the ones which are responsible for the communication between the guest and host ands of the device drivers. These are described as follows:

1. add_buf – adds a new buffer to the queue. The driver had supplied a token which is returned by this function as data once the buffer is consumed.
2. kick – This function is used by the guest driver to kick the host driver and vice versa when buffers have been added to the virtqueues for use.
3. get_buf – It returns a used buffer along with the length that was written to the buffer in its previous usage. This function returns the token handed to add_buf.
4. disable_cb – This function states that the guest wants to disable interrupts signalling use of a buffer.
5. enable_cb – This function restarts the interrupts signalling that the guest is free to process.

6. Network I/O Virtualization

There are two kinds of network backend types provided by the KVM-QEMU hypervisor. One is User networking which is meant for small scale usage of virtualization where the only requirement is access to web pages from the guest machines. The other one is known as Bridged networking which is more suited for a cloud infrastructure as it allows a meaningful participation with the network. It allows both arrival and transmission of traffic and can be used to host a service on the guest machine.

6.1 User Networking

It is implemented using a software called "Slirp" and provides a full TCP/IP stack within QEMU. It uses that stack to implement a virtual NAT'd network with bridges to host network. The guest cannot be directly accessed from the external network. This method is appropriate if the only requirement is user access to internet for browsing web pages. This also does not allow any other protocols except TCP and UDP including ping and ICMP. The slirp layer also acts as a firewall and does not allow the local host to accept arbitrary incoming packets. However, it can utilize port forwarding by binding a host port to a guest port hence, redirecting traffic and allowing file sharing. A brief diagram giving the details is shown in Fig. 22.

The guest OS will see an E1000 NIC (default network adapter emulated by QEMU) with a virtual DHCP server and will be allocated a local IP address. A virtual DNS server will be accessible to the guest, and a virtual SAMBA file server (if present) allowing the user to access files on the host via SAMBA file shares [47]. The external network will see the IP address of the host machine.

A port on guest VM is mapped to a physical port on the guest. On any network traffic exchange, the physical port forwards it to the routing table present in the back end driver which further routes it to the QEMU thread. Inside QEMU, slirp translation layer is used to determine the guest port to which mapping was done and the guest OS is signalled. The guest machines use the DHCP servers present in QEMU to acquire local IP addresses. Theses virtual machines access the host port via a virtual gateway, again emulated by QEMU.

6.2 Bridged Networking

This technique is also known as tap/tun networking. It allows virtual interfaces to connect to the outside network through the physical interfaces, making them appear as normal hosts to the rest of the network. This allows the guest users to host web server applications and hence, is more suitable for a cloud environment. The IP address of the guest machine is its global IP as well which allows it to be a part of the external network.

This method utilizes the bridge utilities of Linux and creates a bridge from the physical NIC to the virtual NIC. A network bridge is a Link Layer device which forwards traffic between

networks based on MAC addresses and is therefore also referred to as a Layer 2 device. It makes forwarding decisions based on tables of MAC addresses which it builds by learning what hosts are connected to each network. A software bridge can be used within a Linux host in order to emulate a hardware bridge, for example in virtualization applications for sharing a NIC with one or more virtual NICs [48]. This method takes advantage of virtio net devices for network I/O and uses DMA support to transfer packets from the receive buffers of physical NIC to the virtqueue buffers.

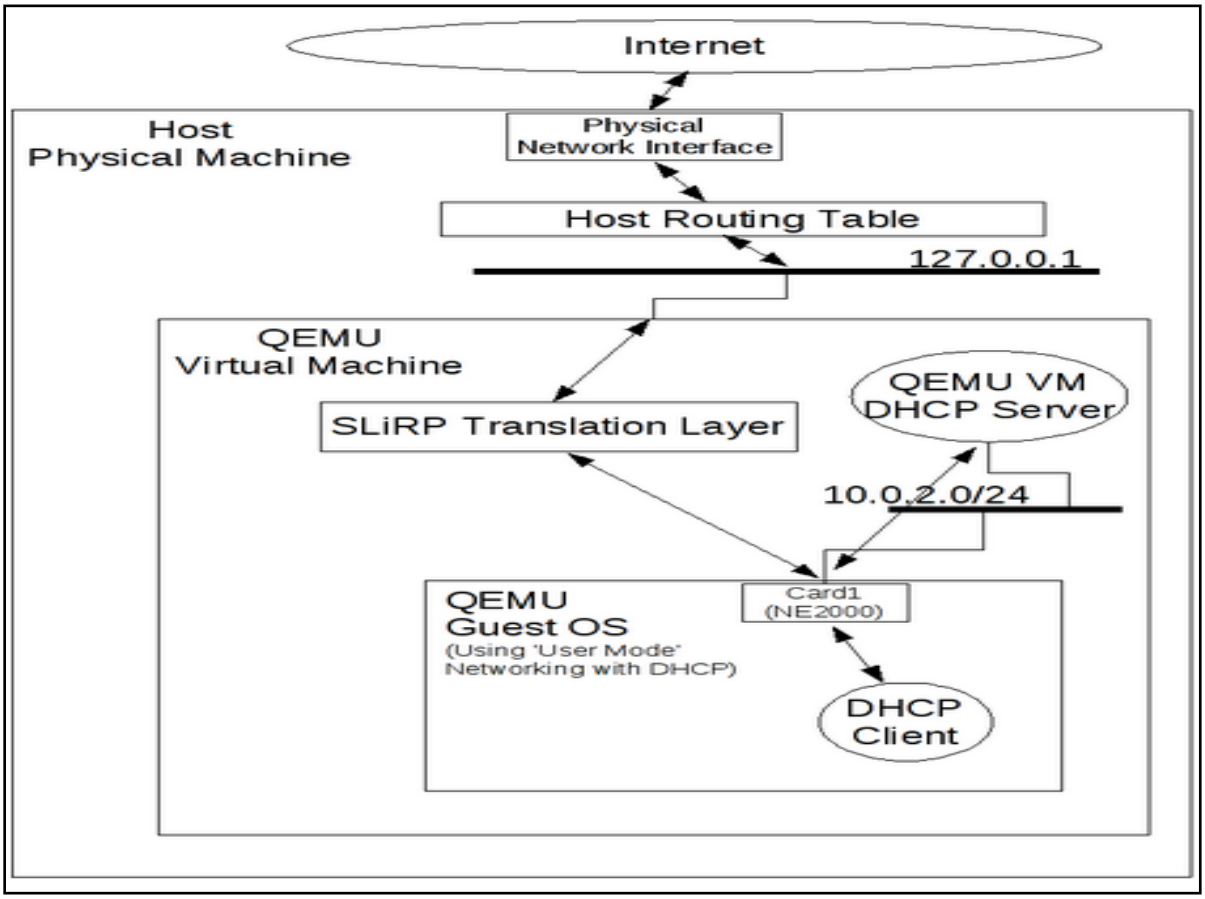
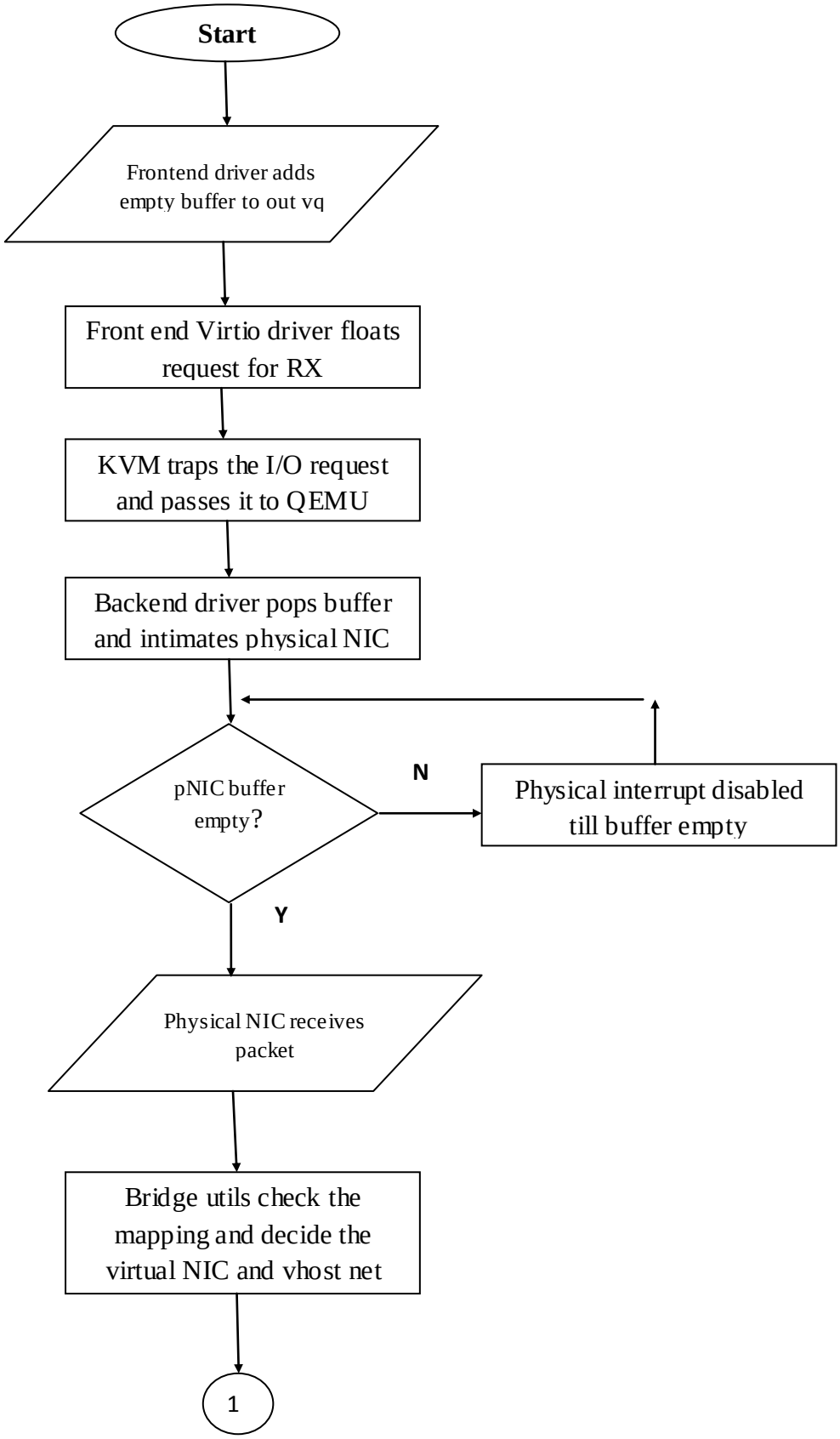


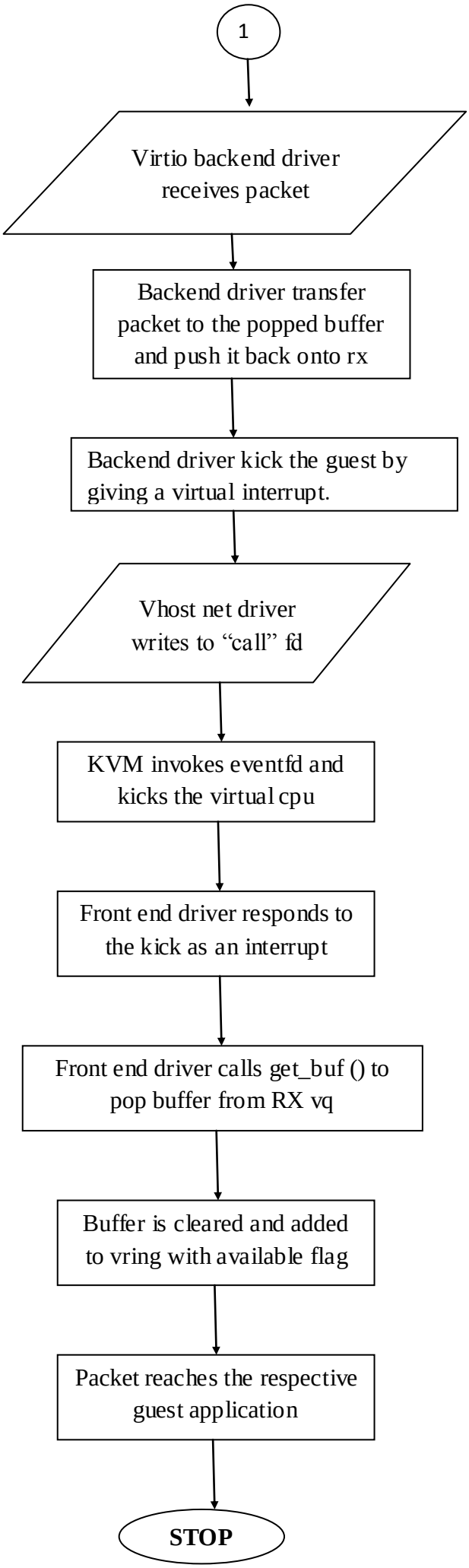
Figure 22 User Networking In QEMU using SLIRP layer [46]

6.3 Networking with Virtio

Vhost net emulates the virtio network card in the host kernel. The vhost – net driver creates a /dev/vhost-net character device on the host. The Vhost net driver is an interface for configuring vhost-net instances [42]. Buffers are placed on the virtqueues by the front end virtio driver and Vhost worker threads receive a kick to handle them. The threads remove the packets from the transmission vq (tx) and forward them to the tap fd. Similarly, when a packet arrives, worker threads collect the packets from tap fd and places them on the receiving vq (rx). It then kicks the guest to receive the packets. The guest usually disables callbacks while transmitting packets as they do not need to know when a packet is transmitted eventually.

A detailed picture of how exchange of packets is handled by KVM is followed in a flowchart. This gives information about the scenario where the guest needs to receive packets from outside.




1
Virtio backend driver receives packet
Backend driver transfer packet to the popped buffer and push it back onto rx
Backend driver kick the guest by giving a virtual interrupt.
Vhost net driver writes to “call” fd
KVM invokes eventfd and kicks the virtual cpu
Front end driver responds to the kick as an interrupt
Front end driver calls get_buf () to pop buffer from RX vq
Buffer is cleared and added to vring with available flag
Packet reaches the respective guest application
STOP

7. Coalescing Algorithm

As is evident from the previous discussions, interrupt handling incurs huge CPU overheads due to the several context switches. This effect is aggrandized even more so when we consider a virtualized environment. It has been estimated that each virtual interrupt can introduce an additional 8000 CPU cycles [28]. Coalescing is the technique using which several interrupts can be clubbed to be delivered in one batch to the guest virtual machine. This reduces the number of context switches, hence reducing the number of CPU cycles and CPU overhead.

Network virtualization can be enhanced for KVM by using multi level coalescing of virtual interrupts to reduce the processor cycles. It can be applied in two layers, one in the backend virtio driver and one in the front end driver in the following ways:

1. Backend Driver – Coalescing of interrupts in this driver, more precisely, waiting till a certain number of packets are received before notifying the guest will control the frequency of interrupt delivery to the guest.
2. Frontend Driver - It does the same thing for the guest OS as a device driver does in non-virtualized scenarios. Applying coalescing to interrupts in this driver (in the same manner as in back end driver) will help in reducing context switches within the guest.

7.1 Proposed Algorithm

The proposed algorithm takes into account two parameters, namely the maximum coalescing parameter and the minimum time interval. Maximum coalescing count signifies the maximum number of interrupts that should be waited for before the guest is notified. Minimum time interval specifies the time limit after which the guest is notified irrespective of the number of packets received. The rate algorithm has been designed such that very low number of interrupts will automatically be adapted to by reducing the rate of coalescing. The suggested time interval is 200 ms as this value has been verified to give good results with controlled latency through experimentation [48].

mC: maximum coalescing count
cC: current number of packets received
mT: minimum time interval after which interrupts are to be delivered
cT: current time period calculated from beginning of current session
begin: timer value at the start of this interval
// the relative timer value i.e. the one pertaining to the guest is used to determine the time elapsed

```
Coalesce ()
{
      cT <-- current_time() - begin;

            if (cC == mC && cT >= mT)
                  cC <-- 0; cT <-- 0;        // Rate remains same as the expected number of
                  virtqueue_kick (vq);       //    interrupts came in expected time


            else if (cC == mC)
                  reviseRate(cC, mC, cT);          // Need to kick the guest just once as
                  cC <-- 0; cT <-- 0;              // once interrupted, it will handle all the
                  virtqueue_kick(vq);              // buffers lying in the RX virtqueue


            else if (cT >= mT)
                  reviseRate (cC, mC, cT);         // Need to kick the guest just once as
                  cC <-- 0; cT <-- 0;              // once interrupted, it will handle all the
                  virtqueue_kick (vq);             // buffers lying in the RX virtqueue

            else
                  cC++; // do not deliver
                  exit;
       exit;

  }
```

Rate: the calculated rate at which interrupts should be delivered

0<Rate≤1; Signifies the number of interrupts to be grouped together.

Nexp: possible number of interrupts with current incoming rate

```
reviseRate (cC, mC, cT)
{
       if (cC == mC)                          // If maximum coalesced number was reached before
             Nexp <-- ceil [(mC * tT) / cT];  // the time limit, there is a suspected increase in the
             Rate <-- 1/ceil [(cC + Nexp)/2]; // activity, Rate should be increased.

      else if (cT >== mT)
             Nexp <-- mC;                     // If minimum time limit was reached before the
             Rate <-- 1/ceil [(cC + Nexp)/2]; // maximum coalesced count, activity is decreased.
                                              // Rate should be decreased.
       mC <-- 1/Rate;

}
```

7.2 Complexity of Algorithm

Space Complexity

The space complexity of the proposed algorithm is O(1) as constant space is occupies by the algorithm.

Time Complexity

The complexity of virtqueue_kick () will be standard as it is a standardized operation. Also, in a scenario where no coalescing is applied, time complexity of KVM in handling network traffic interrupts is equal to the complexity of virtqueue_kick (). Let this be O(K).

Case 1: No coalescing as rate of coalescing is 1.

Complexity of the proposed algorithm is equal to the complexity of virtqueue_kick () function call as no coalescing takes place. Hence, order of complexity is O (K).

Case 2: $cC < mC$ and $cT < mT$

he time complexity of the algorithm is O (1) as no function calls are made. Only the count of interrupts is increased and the function exits.

Case 3: $cC = mC$ or $cT \geq mT$

The time complexity in this depends on two components:

1. virtqueue_kick() – Since a guest takes into account all the buffers in the virtqueue once it is interrupted, this function is called only once. Hence, the complexity becomes O(K).
2. reviseRate() – The time complexity of this function is O(1) as only standard mathematics operations are performed. Complexity of multiplication and division is assumed to be O(1).

Hence, the best case complexity of the proposed algorithm is O(1) while the worst case complexity is O(K), where O(K) is the time complexity of standard interrupt handling function . The average complexity of the algorithm is O (1).

The calling function processes the buffer and calls virtqueue_kick() function for each packet. Earlier, the time complexity of the function was given by [O(size_of_buffer) + O(K)]. After the coalescing algorithm is introduced, the time complexity of calling function is reduced to [O(size_of_buffer)+Rate*O(K)], $0<Rate\leq1$.

7.3 Proof of Correctness

Lemma 1: Number of CPU cycles are reduced by the proposed algorithm.

Proof by contradiction:
Let the number of Interrupts over a time interval be I. Let the number of CPU cycles for one

network interrupt be Ccyc. Let us assume that the proposed algorithm had a non-decreasing effect on interrupt delivery.

In a general scenario with no coalescing applied, number of CPU cycles consumed will be I_no_coalesce = I*Ccyc.

Case1: During high activity phase,
I > mCi, mCi is the maximum coalescing count for i^{th} time interval.
Then,

$mC_{i+1} = [(mCi * mT)\frac{1}{cTi} + cCi]/2$

$I = mC_1 + mC_2 + mC_3 + \ldots\ldots\ldots\ldots + mC_n$

Hence, if I interrupts were handled in n batches, then on an average, I/n interrupts were handled per batch. Since we are considering a high activity phase, I/n>1.

I_coalesce = n*Ccyc

Now, we had assumed that number of CPU cycles did not decrease

That would imply,

$$n * Ccyc \geq I * Ccyc$$
$$\Rightarrow n \geq I \text{ (A contradiction)}$$

Case 2: During low activity phase,
I < mCi, mCi is the maximum coalescing count for i^{th} time interval.
Then,
$I = cC_1 + cC_2 + cC_3 + \ldots\ldots\ldots\ldots + cC_n$
in the worst case, $cC_i = 1$ for each mT.
Then,

Number of CPU cycles consumed will be = $I * Ccyc$ = I_no_coalesce

Hence,

Number of CPU cycles $\leq I * Ccyc$

Average Case:

Let till i^{th} time interval, the frequency of interrupts was high and further it was low till n^{th} time interval.

Then,

$I = mC_1 + \ldots mC_i + cC_{i+1} \ldots + cC_n$

Let $I_1 = mC_1 + \ldots mC_i$ and $I_2 = cC_{i+1} \ldots + cCn$

Now, till i^{th} time interval, number of CPU cycles = i * Ccyc $\leq I_1$* Ccyc (case 1) ------- (1)

Till n^{th} time interval,
number of CPU cycles $\leq$ $[I-(mC_1+..mC_i)]*Ccyc = I_2*Ccyc$. (case 2) -------- (2)

Therefore, taking sum of equation 1 & 2,
$I\_coalesce \leq (I1 + I2) * Ccyc = I*Ccyc$.

Hence, it is proved that the proposed algorithm will reduce CPU over head by reducing the CPU cycles in handling virtual interrupts.

Lemma 2: Number of context switches between the guest and host are reduced.

Proof by Assertion:
Let for each virtual interrupt, let N be the number of context switches that take place during one return trip. Let I be the number of interrupts to be delivered in current pass of the algorithm. A return trip occurs when virtqueue_kick () function call is made.
Then, in case of no coalescing, the number of context switches between guest and host
N_cont_swit = N*I, where I is the number of interrupts.

Case 1: During high activity phase,
virtqueue_kick () is called when mC number of interrupts have been received. It is known that $mC \geq 1$.
Then,

No of context switches = N
No of context switches without coalescing = mC * N
Therefore, decrease in number of context switches = $(mC-1) * N \geq 0$

Case 2: During low activity phase,
virtqueue_kick () is called after mT time has elapsed. The number of coalesced interrupts at this stage is given by cC. It is known that $cC \geq 1$.
Then,

No of context switches = N
No of context switches without coalescing = cC * N
Therefore, decrease in number of context switches = $(cC-1) * N \geq 0$

Every pass of the algorithm will either be a high activity or a low activity phase. Hence, the proposed algorithm decreases the number of context switches between the guest and the host.

8. Real Time Interrupts

A real time operating system is an operating system intended to serve real time application requests. It must be able to process data as it comes in, typically without buffering delays [32]. A real time OS can either have an event driven architecture, wherein, a running task is pre-empted if a higher priority task needs to be serviced or it can be based on giving equal time slices to each task in the ready queue. Basic architecture of a RTOS is as shown in Fig. 23. Non real kernel runs as a process on the real time kernel which schedules the real time processes and non-real time kernel by giving time slices to each. Non real kernel schedules its processes in the time slice provided to it by the real time kernel.

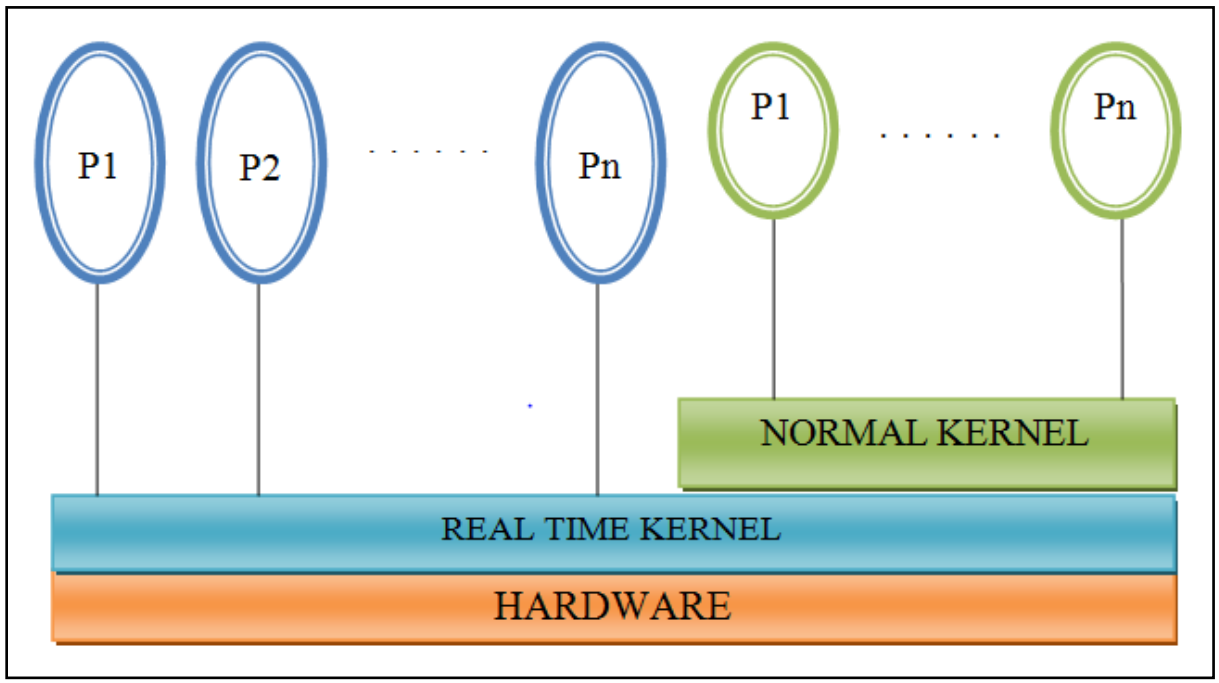


Figure 23 Basic Architecture of Real Time Operating Systems

Soft real time OS is usually able to meet a deadline while a hard real time OS is deterministically able to meet the deadline. Each real time process provides its release time, period and deadline to the kernel. The key features of a real time operating system are minimal interrupt latency and minimal thread switching latency [32]. Communication and resource sharing in a real time operating system is accomplished through temporary disabling of interrupts, semaphores and message passing. Many RTOS allow the application itself to run in kernel mode for getting system call efficiency and also to permit the application to have greater control of operating environment without OS intervention [33].

In case of virtualization, only one of the two scenarios could be possible:

1. The host OS is itself a Real Time Operating System, in which case, none of the guest OSes can be a real time operating systems.
2. One of the guest OSes is a real time operating system, in which case, neither the host nor other guest OSes can be real time operating systems.

Hence, there can be only one real time operating system running on the host machine at a given point of time. Keeping this in mind, three case studies have been enlisted giving possible solutions to incorporate the real time nature of the guest or host OS.

Case 1: Linux as Real Time Operating System with kernel modification

The standard Linux kernel as designed is only able to meet soft real time demands with the help of basic POSIX functionalities provided for time handling. To make Linux kernel adaptive to real time domain, RT-Preempt patch has been introduced. The evolution of real time properties provided by the patch is as shown in Fig. 24. The RT-preempt patch along with a generic clock event layer converts Linux into a fully pre-emptive kernel by introducing the following changes [34]:

1. Making in-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes.
2. Critical sections protected by i.e. spinlock_t and rwlock_t are now preemptible. The creation of non-preemptible sections (in kernel) is still possible with raw_spinlock_t (same APIs like spinlock_t).
3. Implementing priority inheritance for in-kernel spinlocks and semaphores.
4. Converting interrupt handlers into preemptible kernel threads: The RT-Preempt patch treats soft interrupt handlers in kernel thread context, which is represented by a task_struct like a common userspace process. However it is also possible to register an IRQ in kernel context.
5. Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to userspace POSIX timers with high resolution.

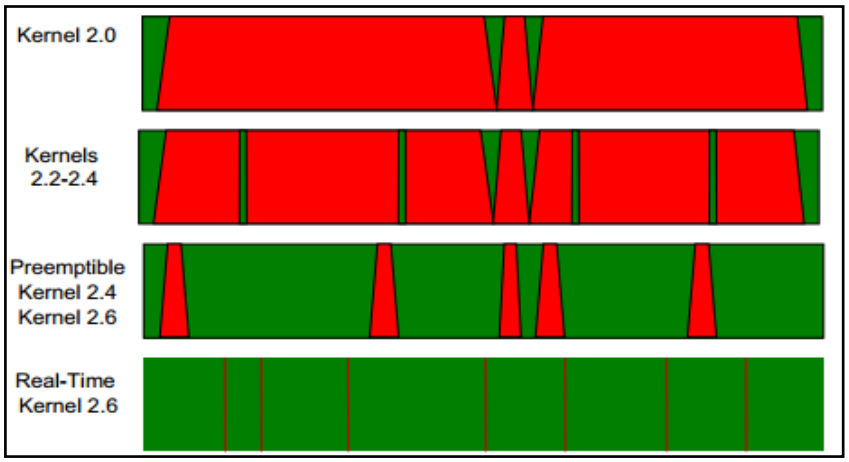


Figure 24 Evolution of real time properties of Linux Kernel using RT_Preempt patch (red – non preemptible, green – preemptible) [35]

Case 2: Linux-KVM as Real Time Hypervisor with Kernel and Guest Modification

Jan Kiszka [36] in his paper on improvising KVM to facilitate acceptable hosting of real time guest operating system has suggested a model for para virtualized scheduling of guest interrupts requests. This model supports three scheduling policies viz. a viz. Fixed time scheduling which is the default scheduling algorithm used by Linux kernel (SCHED_OTHER), First In First Out scheduling algorithm (SCHED_FIFO) and Round Robin scheduling (SCHED_RR).

The hypervisor schedules a virtual CPU according to the scheduling policy and priority reported by the guest when it was last executed. The guest provides a priority called Pguest which is confined to the range [0, 99]. The range of priorities provided to the virtual machines lies between 0 and Pmax, where Pmax is a per virtual CPU priority limit that the hypervisor can reach. Pguest is mapped to host priority according to the following formula:

$$p = \left\lfloor p_{guest} \frac{p_{max}}{99} \right\rfloor$$

The scheduling parameters provided by the guest are applied if neither any virtual interrupts are waiting to be injected into the guest nor are currently being processed by the guest OS. However, if the need arises to inject an interrupt into virtual CPU, its priority is raised to Pmax by the hypervisor. If the scheduling algorithm stated by the guest OS is SCHED_OTHER (default case), it is raised to SCHED_FIFO otherwise it is left unaltered. The priority boost is maintained till the interrupt is serviced. In case of a pending non-maskable interrupt, the priority boosting is maintained further on instead of switching the thread.

The interface for this model requires two new hypercalls to be added to the KVM module and the corresponding call backs to be added to the guest operating systems. The two hypercalls to be added are:

1. Set Scheduling Parameters – This is used to inform the hypervisor about the changes in scheduling policy and priority provided by the guest in case of an interrupt.
2. Interrupt Done – This hypercall is invoked by a virtual CPU when it has finished servicing an interrupt. Since there are no pending interrupts, scheduling parameters are not reset until the next VM entry.

The results for this study showed an increase in the latency by 15% to 25% due to an overhead of the para virtualized scheduling algorithm. However, optimizations by reducing the context switches are in progress. This approach is still under active research.

Case 3: Real Time Scheduling of Virtual Machines without Kernel Modification

This approach was driven by the need of a scheduling model that could satisfy the real time requirements of guest applications but without any modifications to the KVM module. Mikael Asberg et al. [37] proposed a framework scheduling guest applications using Hierarchical Fixed Priority Scheduling without kernel modification. This was accomplished by developing a loadable kernel module called RESCH [38] which can be integrated as and when the need arises.

The hierarchical scheduling framework used HFPS to schedule real time Linux tasks in groups. The interface for this framework took budget, priority and period as parameters for each group. The groups were scheduled based on their priority for budget time units after every period. This approach utilized nested scheduling where the same procedure was followed within each group of processes as well. The basic architecture of Linux integrated with RESCH is shown in Fig. 25.

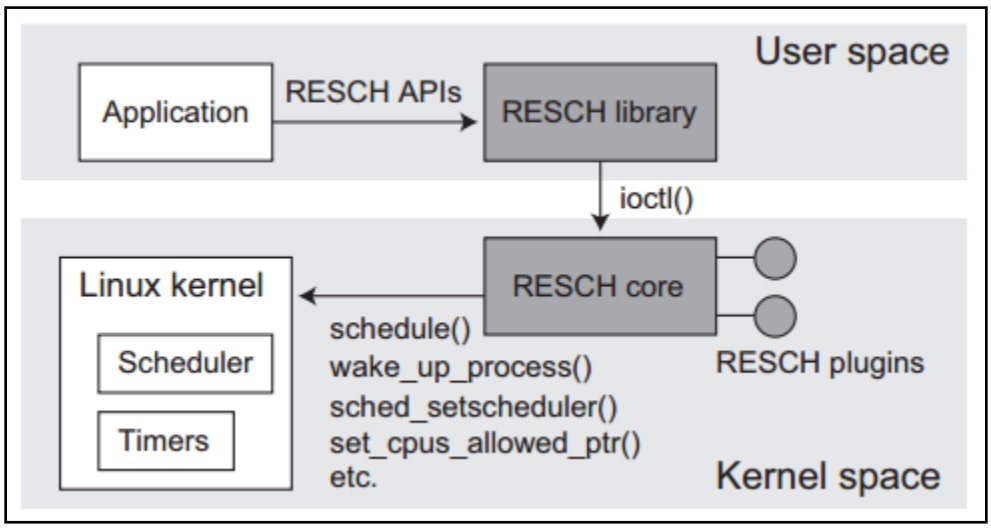


Figure 25 Conceptual Framework of RESCH[38]

Since the virtual machines are like applications for the underlying Linux kernel, they are linked to the RESCH APIs present in kernel during compilation. The applications communicate with RESCH through ioctl() commands. RESCH uses Completely Fair Scheduler present in Linux kernel to isolate real and normal processes. Real time processes are always executed before normal processes.

Basic APIs provided by RESCH architecture include [38]:

1. To determine a caller as real time task.
2. To change the caller's status back to normal after interrupt has been handled.
3. To wait for next period by sleeping with TASK_UNINTERRUPTIBLE status. This method returns failure if the deadline is missed.
4. To set worst case execution time of caller.
5. To set relative deadline of caller.

6. To set minimum inter arrival time of caller.
7. To set priority of a caller.

Though RESCH comes out as the better choice of the three case studies, there are some associated disadvantages with this technique. Since RESCH is a loadable kernel module, there are some inherent memory and processor overheads. It is loaded into the kernel whenever a real time task is to be scheduled and is removed if there is a need to free memory or it is no longer required.

There are other known methods as well to realize real time requirements of guest OSes. One of the most straightforward approaches is to increase the priority of the QEMU threads associated with each virtual CPU. However, it might result in starvation of host processes which are being sidelined by guest. Another measure to mitigate the risk is by preventing over commitment of CPU resources. Care can be taken to accommodate lesser virtual CPUs than the real processer cores. But this will not be an ideal scenario for cloud providers who aim for as extensive multiplexing of resources as is possible. Another approach is for the hypervisors to give higher priority to the real time virtual machine over other guests. This technique will be appropriate if the real time guest executes only important processes otherwise it might result in starvation of the processes of other virtual machines. This priority inversion can be overruled if the virtual machine provides the hypervisor with information about its internal state. However, this can be achieved only if the guest OS is modified which is not optimum.

The method which can be used in the proposed algorithm to mitigate the interrupt latency for real time tasks is to program the timer in an optimum way. If the interrupt service completion message is delivered to the concerned virtual CPU in a reasonable interval of time, the applications will not suffer much from the latency.

9. Conclusion

This report shows that I/O virtualization is an inextricable part of virtualization and acts as a major bottle-neck. Network I/O virtualization forms a major part of I/O virtualization and the interrupts due to network packets induce large overheads. In a virtual environment, where there is a demand for hosting as many virtual machines on a physical machine as is possible, this overhead increases manifolds and causes severe latency in performance and low throughput.

KVM uses the functionalities of QEMU and Virtio for emulating the I/O requests of the guests. For each interrupt that is generated in a virtual machine, KVM causes a VM exit, handles the interrupt and then emulates its completion in the guest VM which causes a number of context switches. For network I/O, arrival or transmission of each packet induces a VMexit. These transfers are taken care of by the virtio-net drivers which kick the guest/host on subsequent action.

Coalescing these interrupts for guests results in a positive impact on the performance of the virtual machine. The complexity of the calling function is reduced. The proposed algorithm reduces the number of context switches and results in lesser CPU overhead. Since the worst case time and space complexities do not vary much from that of the existing hypervisor code, the algorithm does not incur CPU/memory overhead. Multi level coalescing will further improve the efficiency of the algorithm.

Work Done by Student

The thesis was initiated with gaining an in-depth knowledge of the novel concepts of cloud computing and virtualization. The open problems in this field were then recognized and explored. After settling on reducing CPU overhead in handling virtual interrupts, existing techniques for the same were studied. Following this, the existing hypervisors were probed and KVM was chosen based on the facts that it was an open source software with Linux kernel support and it used the primitive method for handling virtual interrupts. The internals of KVM, QEMU and Virtio were then studied basing the study on x86 Intel architecture. Since, x86 provides hardware assistance, focus was directed to specific device drivers. **Contribution to Research** - An algorithm was proposed for effective interrupt handling using the technique of coalescing for network I/O drivers. To enhance the suggested approach, real time interrupts were also considered. Different scenarios were considered and appropriate methodologies were studied for each that users can implement. Though, keeping an optimum timer limit was considered as a solution in current algorithm.

Future Scope

The proposed algorithm can be integrated with the KVM module and experiments can be conducted to verify the results. The algorithm to generate rate of coalescing can be made more adaptive by considering the rate of arrival of interrupts over a larger period of time than just the previous.

References

[1] en.wikipedia.org/wiki/Virtualization

[2] Yaser Ghanam, Jennifer Ferreira, Frank Maurer. (2012) Emerging Issues & Challenges in Cloud Computing— A Hybrid Approach. In Journal *of Software Engineering and Applications*. [Online] 5, pp.923-937.

[3]Gordon,A., Amit,N. ,Har'El,N., Ben-Yehuda, M., Landau,A., Schuster,A., Tsafrir,D. (2012*)* ELI:Bare-Metal Performance for I/O Virtualization. *In ACM SIGARC H* – dl.ac m.org

[4] Hwang, K., Fox, G.C. & Dongarra, J.J. (2012) *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Singapore. Morgan Kauffman

[5] Waldspurger, C., Osenbelum, M. (2012) I/O Virtualization. *In Communications of the ACM* [Online]. Vol. 55, Issue. 1, pp.66-73. Available at : http://cacm.ac m.org/ magazines/2012/1/ 144808-i-o-virtualizat ion/fulltext

[6] Mahalingam,M ., (VMware 2006*) I/O architectures for Virtualization*. Available at : xpgc.vicp.net/course/svt/TechDoc

[7] en.wikipedia .org/wiki/Interrupt

[8] Dong,Y., Dai,J., Huang,Z., Guant,H., Tian,K., Jiang,Y.,(2009) Towards High-Quality I/O Virtualization. In *SYSTOR'09*.4-6 May 2009. Haifa.
Available at: http://140.115.52.150/ 98ht ml/paper/pdf/Towards%20High-Quality% 20IO% 20Virtualizat ion.pdf

[9] Ben-Yehuda, M., Day, M. D., Dubitzky, Z.,Factor, M.,Har'el, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour,B.A. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX Symposium on Operating Systems Design & Implementation* (OSDI) (2010).

[10] Liao, G., Guo, D., Bhuyan, L., And King, S. R. (2008) Software techniques to improve virtualized I/O performance on multi-core systems. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*

[11] Liu,J., Abali,B. (2009) Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization. In *ICS'09*. New York. 8-12 June 2009. ACM.

[12] Ah mad, I., Gulati, A., And Mashtizadeh, A. (2011) vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *USENIX Annual Technical Conference* (ATC)

[13] Kivity,A., Kamay,Y., Labor,D. (2007) kvm: the Linux Virtual Machine Monitor. 2007 Linux Symposium.Ottawa. *Proceedings of Linux Symposium.* Vol. 1. 27-30 June, 2007.pp. 225-230

[14] QumranetInc., (2006). White Paper on KVM: Kernel based Virtualization Driver.
Available at: http://www.linu xinsight.com/files/kvm_ whitepaper.pdf

[15] Goto, Y., (2011) Kernel-based Virtual Machine Technology. In FUJITSU *Science Technology Journal*, Vol. 47, Issue. 3, pp. 362-368.

[16]Betak, T., Duley, A., and Angepat, H. Reflective Virtualization Improving The Performance Of Fully-Virtualized X86 Operating Systems.
Available at : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.7868

[17] Dovrolis, C., Thayer, B., And Ramanathan, P. (2001) HIP: Hybrid Interrupt-Polling For The Network Interface. In ACM *SIGOPS Operating Systems Review* (OSR) .Vol. 35, pp. 50–60.

[18] Armbrust, M., Fox, A., Griffith, R., et. Al.[2009] Above the Clouds: A Berkeley View of Cloud Computing. [Online] Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

[19] Ali, I.,Maghanathan, N.,Virtual Machines and Networks- Installation, Performance, Study, Advantages and Virtualization Options. In *International Journal of Network Security & Its Applications (IJNSA)*. Vol. 3.January, 2011.
[20] http://en.m.wikipedia.org/wiki/Ring_(computer_security)
[21] RedHat, Inc., [2009]. KVM – Kernel Based Virtual Machine. Available at:
http://www.redhat.com/rhecm/rest-rhecm/jcr/repository/collaboration/ jcr:system/
jcr:versionStorage/5e7884ed7f00000102c317385572f1b1/1/jcr:frozenNode/rh:pdfFile.pdf
[22] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, Yeh-Ching Chung, *ARMvisor: System Virtualization for ARM*, *Linux Symposium*
[23] VMware Inc., vSphere ESX 5 Documentation Center [Online]. Available at:
http://pubs.vmware.com/vsphere-50/index.jsp?topic=%2Fcom.vmware.vsphere.resmgmt.doc_
50%2FGUID-B14C8267-C2A4-4BF8-B680-70C2B350B325.html
[24] Intel Virtualization Technology for Directed I/O. Rev. 2.2. Intel Corporation, September 2013
[25] Devine,S., (VMware) Virtual Mchines: Memory Virtualization.[Online]
Available at : xpgc.vicp.net/course/svt/TechDoc
[26] VMware Inc., (2007). White Paper on KVMUnderstanding Full Virtualization, Para Virtualization, and Hardware Assist.
Available at: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf
[27] Waldspurger,C., Osenbelum,M. (Nov.,2011) I/O Virtualization. *In Communications of the ACM* [Online]. Vol. 55, Issue. 1, pp.66-73. Available at : http://queue.acm.org/detail.cfm?id=2071256
[28] Dong, Y., Xu, D., Zhang, Y., Liao, G., [2011] Optimizing Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive Side Scaling. In *IEEE International Conference on Cluster Computing.*
[29] en.wikipedia.org/wiki/Device_driver
[30] Corbet, J., Rubini, A., Kroah-Hartman, G., Linux Device Drivers. 3rd Edition. USA. O'Reilly, 2005
[31] Jones, T., [IBM, 2010] "An introduction to bottom halves in Linux 2.6" in Kernel APIs, Part 2: Deferrable functions, kernel tasklets and work queues.[Online]
Available at: http://www.ibm.com/developerworks/library/l-tasklets/
[32] en.wikipedia.org/wiki/Real-time_operating_system
[33] Giovanni [ChibiOS,2003] Response Time and Jitter.[Online]
Available at : http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:jitter
[34] http://elinux.org/images/e/ef/InterruptThreads-Slides_Anderson.pdf
[35] http://lxr.free-electrons.com/source/drivers/virtio/
[36] Kiszka, J.[Siemens AG, Munich], Towards Linux as Real-Time Hypervisor. [Online]
Available at: http://old.lwn.net/lwn/images/conf/rtlws11/papers/proc/p18.pdf
[37] Asberg,M., Forsberg, N., Nolte, T., Kato, S. Towards Real-Time Scheduling of Virtual Machines without Kernel Modifications.[Online]
Available at: http://www.ipr.mdh.se/pdf_publications/2173.pdf
[38] Kato, S., Rajkumar, R., Ishikawa, Y. A Loadable Real-Time Scheduler Suite for Multicore Platforms [Online] Available at: http://ertl.jp/~shinpei/papers/techrep09.pdf
[39] Che, X., Wang, H., Lu, H., Yao,P. [2013] Research of KVM Interrupted and Virtual Clock presented in 2nd International Conference on *Intelligent System and Applied Material (GSAM 2013)*
[40] Nahum, University of Columbia, Coursework on Network System Design and Implantation [2010], Presentation on Xen KVM Networking

[41] http://en.wikipedia.org/wiki/QEMU#KVM
[42] Hajnoczi,S.,[RedHat Inc., 2011] QEMU Internals: Overall architecture and threading Model [Online]. Available at: http:// vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html
[43] Russell, R., [IBM OzLabs]. Virtio: Towards a De-Facto Standard for Virtual I/O Devices [Online] Available at: http://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf
[44] Jones,T., [IBM, 2010] Virtio: An I/O virtualization framework for Linux [Online]
Available at: http://www.ibm.com/developerworks/linux/library/l-virtio/index.html?ca=dgr-lnxw97Viriodth-LX&S_TACT=105AGX59&S_CMP=grlnxw97
[45] Hajnoczi, S., [RedHat Inc., 2011] QEMU Internals: Vhost Architecture [Online]. Available at: http:// vmsplice.net/2011/03/qemu-internals-vhost-architecture-and.html
[46] http://bsdwiki.reedmedia.net/wiki/networking_qemu_virtual_bsd_systems.html
[47] http://en.wikibooks.org/wiki/QEMU/Networking
[48] Ahmad, I., Austruy, M., Mahalingam, M. [VMware, Inc.], Interrupt Coalescing for outstanding Input/Output Completions, US Patent 20130297832 A1, Nov. 7, 2013
[49] GitHub: QEMU Source code, Available at: http://git.qemu.org/qemu.git
[50] GitHub: KVM Source Code, Available at: http://www.linux-kvm.org/page/Code