Performance and Energy Efficiency of Virtual Machine Based Clouds

by

Ryan William Shea

B.Sc., Simon Fraser University, 2010

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

© Ryan William Shea 2016
SIMON FRASER UNIVERSITY
Spring 2016

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name:	**Ryan William Shea**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	***Performance and Energy Efficiency of Virtual Machine Based Clouds***
Examining Committee:	**Chair:** Fred Popowich Professor

Jiangchuan Liu
Senior Supervisor
University Professor _____

Ke Wang
Supervisor
Professor _____

Arrvindh Shriraman
Internal Examiner
Assistant Professor
School of Computing Science _____

Shervin Shirmohammadi
External Examiner
Professor
School of Electrical Engineering
and Computer Science
University of Ottawa _____

Date Defended: 5 January 2016

Abstract

Combining high-speed network accesses and powerful computer virtualization, cloud computing provides an elastic and cost-effective service paradigm for the development of resource-hungry new applications as well as the expansion of a broad spectrum of existing applications. Virtualization has become wide spread in modern computers and operating systems; the extensive use of *virtual machines* (VMs) in a networked cloud environment, however, comes with unprecedented overhead. In this thesis we study the impact of virtualization overhead on real-world systems. First, we present a study on the performance of modern virtualization solutions under DoS attacks. We experiment with the full spectrum of modern virtualization techniques, from paravirtualization, hardware virtualization, to container virtualization, with a comprehensive set of benchmarks. Our results reveal severe vulnerability of modern virtualization. Further, this thesis presents an empirical study on the power consumption of typical virtualization packages while performing network tasks. We find that both Hardware Virtualization and Paravirtualization add considerable energy overhead, affecting both sending and receiving, and a busy virtualized web-server may consume 40% more energy than its non-virtualized counterparts. Next, we focused on cloud network performance instability in the public cloud. Through measurement of real world cloud platforms, we find that network performance degradation and variation phenomena can be prevalent and significant with both TCP and UDP traffic, even within the same data center, and even with a lightly utilized network. Our in-depth measurement and detailed system analysis reveal that the performance variation and degradation are mainly due to the requirements of the CPU in both computation and network communication. Finally, we study the use of virtual machine based clouds to provide infrastructure to support cloud gaming. For each issue we further suggest solutions to resolve performance issues associated with virtualized environments, both at the hypervisor level and, inside a VM, at the operating system level. Our implementation on both large public clouds and our cloud testbed have demonstrated their practicality as well as their effectiveness in improving and stabilizing the energy consumption and performance of virtual machine based clouds.

Keywords: Virtualization; Networking; Cloud Computing; Network Interface Cards; NIC; Graphics Processing Unit; GPU; Energy Efficiency

Acknowledgements

First, I acknowledge my family for their years of support and encouragement.

Also, I would like to thank my supervisor Professor Jiangchuan Liu, your help and guidance was instrumental in the writing of this thesis.

Last but certainly not least, I acknowledge my wife Jane. The years of love, patience, and support you have shown me has made this all possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background

From remote login and server consolidation, to cloud computing, *computer virtualization* has fast become a indispensable technology in today's modern information technology infrastructure. In its simplest form, computer virtualization can be described as the ability to host many *Virtual Machines* (VMs) on a single physical machine. Surveys have shown that 90% of large organizations use computer virtualization in some capacity in their IT infrastructure [10]. The rapid uptake of this technology has been fueled by promises of increased resource utilization, server consolidation, and increased flexibility [67]. It has also made the long held dream of public utility computing platforms a reality. One prominent utility computing example is Amazon's EC2 which went public in 2006 and is now estimated to generate 500 million dollars in revenue per year.

It is now clear computer virtualization has been a success and is here to stay. Yet as with any new technology, there are pros and cons, and computer virtualization is no exception. Running multiple VMs on a single physical machine inevitably creates significant overhead and, not surprisingly, almost every computer subsystem is affected to some degree. The networking interfaces and graphical processing unit (GPU), however, can suffer some of the worst overhead given its complicated operations and interactions with other subsystems. This overhead is defined as the computational and energy cost of maintaining these virtualized environment.

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

Further, as mentioned previously GPUs can also suffer from unprecedented overhead in virtualized environments. Although, GPUs cards have been virtualized to some degree in modern virtualization systems, their performance has generally been poor [15]. This

is because the high-memory bandwidth demands and difficultly in multi-tasking make a virtualized GPU a poor performer when multiple VMs attempt to share a single GPU. However, recent advances in terms of both hardware design and virtualization software design have allowed virtual machines to directly access physical GPUs and exploit the hardware's acceleration features. This access is provided using hardware, and grants a single VM a one-to-one hardware mapping between itself and the GPU. These advances have allowed the cloud platforms to offer virtual machine instances with GPU capabilities. For example, Amazon EC2 has added a new instance class know as "GPU instances", which have dedicated Nvidia Tesla or GRID controllers for GPU accelerated computing.

1.1 Contributions of this thesis

In each chapter of this thesis we perform a comprehensive study on the widely deployed virtualization systems found in modern IT architectures. Using real world test-beds we analyze the performance of virtualization systems running different work loads. Further, we offer optimizations and improvements to these systems and provide practical implementations.

In **Chapter 1** we show that the network interface and GPU are among the most difficult computer subsystems to be virtualized. This chapter also provides a background on both the technology and research pertinent to this thesis. Through both system analysis and real world experiments we motivate the effect virtualization can have on different subsystems.

Chapter 2 presents the first study on the performance of modern virtualization solutions under networked Denial of Service (DoS) attacks. We devise a representative set of experiments to examine the performance of most typical virtualization techniques under standard TCP-based Distributed Denial of Service (DDoS) attacks. We also compare them with the same DDoS on the same services running on non-virtualized servers. Our experiments cover a full spectrum of virtualization solutions with state-of-the-art implementations, and we also examine a comprehensive set of benchmarks. The work in this chapter were published in 2012 in IEEE/ACM IWQoS [52] and was awarded the best student paper award.

Chapter 3 presents an empirical study on the power consumption of typical computer virtualization packages, including KVM, Xen and OpenVZ, while the systems are performing network tasks. We find that both Hardware Virtualization and Paravirtualization systems add a considerable amount of energy overhead to networking tasks. Both TCP sending and receiving can be noticeably affected, and a busy virtualized web-server may consume up to 40% more energy than its non-virtualized counterparts. We have conducted detailed profiling to analyze the workflow for packet delivery in virtualized machines, which reveals that a VM can take nearly 5 times more cycles to deliver a packet than a bare-metal machine, and is also much less efficient on the system's hardware caches. The existence of hypervisors in VMs can dramatically increase interrupts and memory accesses, which in

turn leads to significantly more power consumption for network transactions than a bare-metal machine does. The work was published in 2014 in the proceedings of the 33rd Annual IEEE International Conference on Computer Communications (INFOCOM'14),[62]

Chapter 4 for the first time systematically investigates virtualization overhead on network performance in VM based cloud platforms, striving to not only understand the performance degradation and variation phenomena clearly but also dive deeply to uncover the root causes. To this end, we take the Xen hypervisor based cloud Amazon EC2 as a case study, conducting extensive measurements on the network performance. Our measurement results show that the network performance degradation and variation phenomena can be prevalent and significant even within the same data center. The work was published in 2014 in the proceedings of the 33rd Annual IEEE International Conference on Computer Communications (INFOCOM'14) [61].

Chapter 5 closely examines the performance of modern virtualization systems equipped with virtualized GPU and pass-through techniques. Our results showed virtualization for GPUs has greatly improved and is ready for gaming over a public virtualized cloud. A modern platform can host three concurrent gaming sessions running inside different VMs at 95% of the optimal performance of a bare-metal counterpart. Although there is degradation of memory transfer between a virtualized system's main memory and its assigned GPU, the game performance at the full HD resolution of 1080p was only marginally impacted. The work presented in this chapter were published in 2015 in IEEE Transactions on Circuits and Systems for Video Technology [59]

Chapter 6 concludes the thesis and discusses future research directions.

Next, we introduce computer virtualization and give a short history.

1.2 Virtualization: An Overview

Any virtualization system must ensure that each Virtual Machine (VM) be given fair and secure access to the underlying hardware. In state-of-the-art virtualization systems, this is often achieved through the use of a software module known as a *hypervisor*, which works as an arbiter between a VM's virtual devices and the underlying physical devices. The use of a hypervisor brings many advantages, such as device sharing, performance isolation, and security between running VMs. Having to consult the hypervisor each time a VM makes a privileged call however introduces considerable overhead as the hypervisor must be brought online to process each request. Figure 1.1 shows the interactions between three different VMs and the hypervisor. The solutions to computer virtualization can be broadly split into three categories as described below.

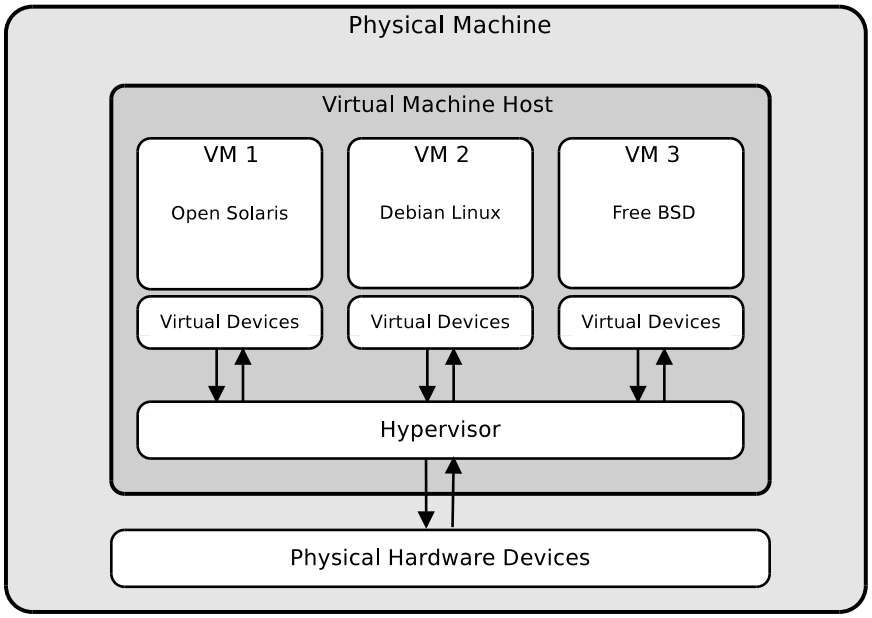
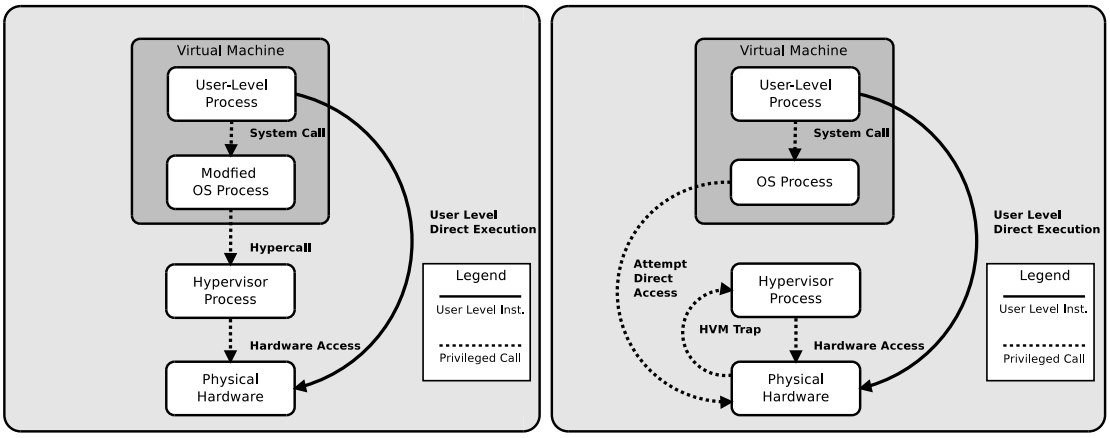


Figure 1.1: Hypervisor based computer virtualization



(a) PVM (b) HVM

Figure 1.2: PVM and HVM simplified architecture

1.2.1 Paravirtualization Machines (PVM)

Paravirtualization Machines, often referred to as PVM, were the first form of full computer virtualization and are still widely deployed today. The roots of paravirtualization run very deep indeed with the first production system know as the VM/370[1] [13]. VM/370 was created by IBM and became available in 1972, many years before paravirtualization became a mainstream product. The VM/370 was a multi-user system which provided the illusion that each user had their own operating system. Paravirtualization requires no special hardware and is implemented by modifications to the VM's operating system. The modifications instruct the OS to access hardware and make privileged system calls through the hypervisor; any attempt to circumvent the hypervisor will result in the request being denied. Modifying the OS does not usually create a barrier for open source operating systems such as Linux; however proprietary operating systems such as Microsoft Windows can pose a considerable challenge. The flagship example of a PVM system is Xen[2], which first became an open source project in 2002. The Xen Hypervisor is also a keystone technology in Amazon's successful cloud service EC2.

1.2.2 Hardware Virtual Machines (HVM)

Hardware Virtual Machines (HVM) rely on special hardware to achieve computer virtualization. HVM works by intercepting privileged calls from a Virtual Machine and handing these calls to the hypervisor. The hypervisor decides how to handle the call, ensuring security, fairness and isolation between running VMs. The use of hardware to trap privileged calls from the VMs allows multiple unmodified operating systems to run. This provides tremendous flexibility as system administrators can now run both proprietary and legacy operating systems in the virtual machine. In 2005, the first HVM compatible CPU became available and as of 2012 nearly all server-class and most desktop-class CPUs support HVM extensions. Both Intel and AMD implement HVM extensions, referred to as Intel VT-X, and AMD-V, respectively. This creates great flexibility for the guest since practically any OS can be run in these VMs. It has been noticed however that HVMs can also have the highest virtualization overhead and as such may not always be the best choice for a particular situation [49][72]. Since HVM must intercept each privileged call, considerably higher overhead can be experienced when compared to PVM. This overhead can be especially high when dealing with I/O devices such as the network card, a problem which has lead to the creation of paravirtualization drivers. Paravirtualization drivers, such as VirtIO package for KVM, allow a VM to reap the benefits of HVM such as an unmodified operating system

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

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

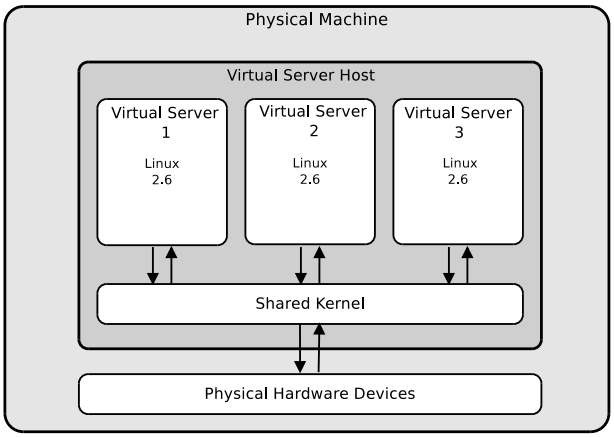


Figure 1.3: Container Virtualization

while mitigating much of the overhead [56]. Examples of HVM-based virtualization systems include KVM[3] and VMware[4] server.

Figure 1.2 shows the simplified architecture of both hardware virtualization and paravirtualization. It is important to note that PVM and HVM are not mutually exclusive and often overlap, borrowing aspects from each other. For example the PVM hypervisor Xen can use hardware virtualization extensions to host proprietary operating systems such as Microsoft Windows. The use of the VirtIO paravirtualization drivers by KVM is an example of HVM using a hybrid approach to virtualization.

1.2.3 Container Virtualization

Container Virtualization, also known as OS-level virtualization, creates multiple secure containers to run different applications in. It is based on the intuition that a server administrator may wish to isolate different applications for security or performance reasons while maintaining the same OS across each container. Container virtualization allows a user to share a single kernel between multiple containers and have them securely use computer resources with minimal interference from others containers. It has been shown to have the lowest overhead among all the existing virtualization techniques [49]. This superiority however comes at the price of much less flexibility as compared to other solutions. In short, the user cannot mix different operating systems, e.g., a Debian Squeeze and an Open Solaris. Typical container virtualization implementations include OpenVZ, Linux-VServer and Solaris Zones. Figure 1.3 shows how containers interact with their shared kernel.

It is important to note that Hardware Virtualization and Paravirtualization both use a *Hypervisor* to interact with the underlying hardware, whereas Container Virtualization

[3] `http://www.linux-kvm.org/`
[4] `http://www.vmware.com/`

does not. This distinction is crucial because the hypervisor acting as a gatekeeper to the underlying hardware generally improves performance isolation between guests on a host.

1.3 Virtualizing Devices

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

1.3.1 Virtualizing the Network Interface Card (NIC)

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

Network interface virtualization, on the other hand, still lacks a rich and well developed set of techniques that allow many processes to securely and directly access the device with performance or security guarantee. Through the use of IP addresses and port numbers, we have to rely on the many layers of the network stack to keep processes from interfering with each other. With virtualization, we have duplication in the network stack, and a

packet must travel through this duplication which can cause a single packet to traverse many different processes before reaching its destination.

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

1.4 Virtual Network Interfaces: Implementation and Performance

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

1.4.1 Software-based Virtual Interfaces

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

For this reason, the emulated interfaces are only used if absolutely necessary and are rarely seen in high performance applications.

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

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

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

We compiled KVM 0.15.1 from source and installed it in our test system. Our test system was a modern mid-range PC with an Intel Core 2 Q9500 quad core processor running at 2.83 Ghz. We enabled Intel VT-X in the BIOS as it is required for Hardware Virtual Machines (HVM) support. The PC was equipped with 4 GB of 1333 MHz DDR-3 SDRAM and a 320 GB 7200 RPM hard drive with 16MB cache. The physical network interface is a 1000 Mb/s Broadcom Ethernet adapter attached to the PCI-E bus. The host and KVM guest used Debian Squeeze as their operating system. We used the `Iperf`[5] network benchmark to create the TCP traffic to and from a remote host in the same subnet. We limit the remote host to 10 Mb/s using the Ethernet configuration tool `Ethtool`. We use the Linux hardware performance analysis tool `Perf`[6] to collect system level statistics such as processor cycles consumed, last level cache references, interrupt requests and processor context switches. For each experiment, we instruct `Perf` to collect five samples each with a duration of 10 seconds and then average them. We test 3 systems: our bare metal Linux host, KVM with paravirtualized VirtIO drivers, and KVM with an emulated rtl8139 network card. It is important to note that, although we use KVM for our experiments, we expect similar results would be found with any virtualization package that relies on software based virtual network interfaces.

[5] `https://iperf.fr/`

[6] `https://perf.wiki.kernel.org`

	Bare-Metal	KVM VirtIO	KVM rtl8139
Cycles	11.5M/Sec	51.8M/Sec	95.6M/Sec
LLC References	0.48M/Sec	2.3M/Sec	3.6M/Sec
Context Switches	189/Sec	452/Sec	452/Sec
IRQs	600/Sec	2600/Sec	5000/Sec

Table 1.1: Iperf receiving 10 Mb/s TCP traffic

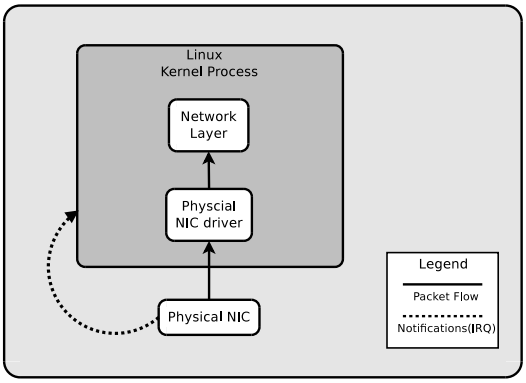
	Bare-Metal	KVM VirtIO	KVM rtl8139
Cycles	4.1M/Sec	33.4M/Sec	74.2M/Sec
LLC References	0.13M/Sec	1.3M/Sec	2.7M/Sec
Context Switches	19/Sec	307/Sec	432/Sec
IRQs	180/Sec	1700/Sec	3500/Sec

Table 1.2: Iperf sending 10 Mb/s TCP traffic

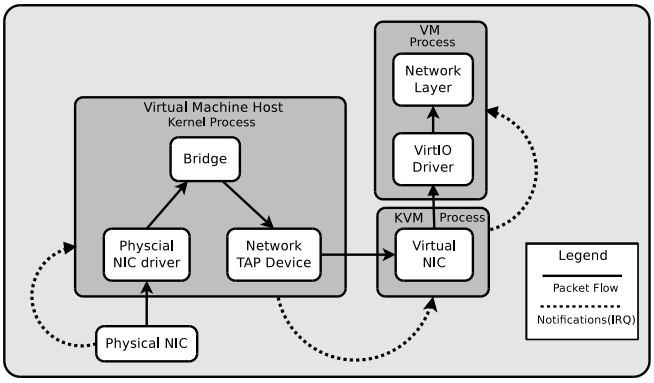
The results for the receiving experiment are given in Table 1.1 and the results for the sending experiment are given in Table 1.2. The CPU cycles and LLC references are given in millions per second. The context switches and interrupt requests are given in number per second. In the receiving experiment, KVM with VirtIO takes nearly 5 times more cycles to deliver the packets to the VM as the bare-metal host. This however is a huge improvement over the emulated rtl8139 interface, which takes nearly 9 times more cycles to deliver the same amount of traffic to the VM. KVM with VirtIO and the emulated rtl8139 are also much less efficient on cache than the bare-metal system. This is due to the fact that the VMs hypervisor must copy each packet from the memory space of the host to the VMs space. These copies use up valuable processor cycles and can also evict data from the processor cache. VirtIO manages to be much more efficient than the emulated device because it generally copies more packets sequentially. The number of context switches increases for both the paravirtualized and emulated interfaces, because the hypervisor must be brought on a CPU in order to copy the packet into the VMs memory space. Finally, we look at IRQs that are used by the physical device to notify the kernel of an incoming packet, and by the hypervisor to indicate to a running VM that it has received packets. Since KVM also uses interrupts to indicate to a running VM that it has received packets from the network, it comes as no surprise that they would be considerably higher. In comparison to the emulated rtl8139 interface, VirtIO manages to reduce the IRQs by generally delivering more packets with a single interrupt.

We now focus on the sending system level statistics found in Table 1.2. As can be seen, all metrics have decreased, which is unsurprising considering sending is a more deterministic task from the point of view of the sender, as it has more control over the data flow.

The reason that software virtual interfaces have several times the overhead of bare-metal interfaces can be explained by comparing the packet and interrupt path in the system.

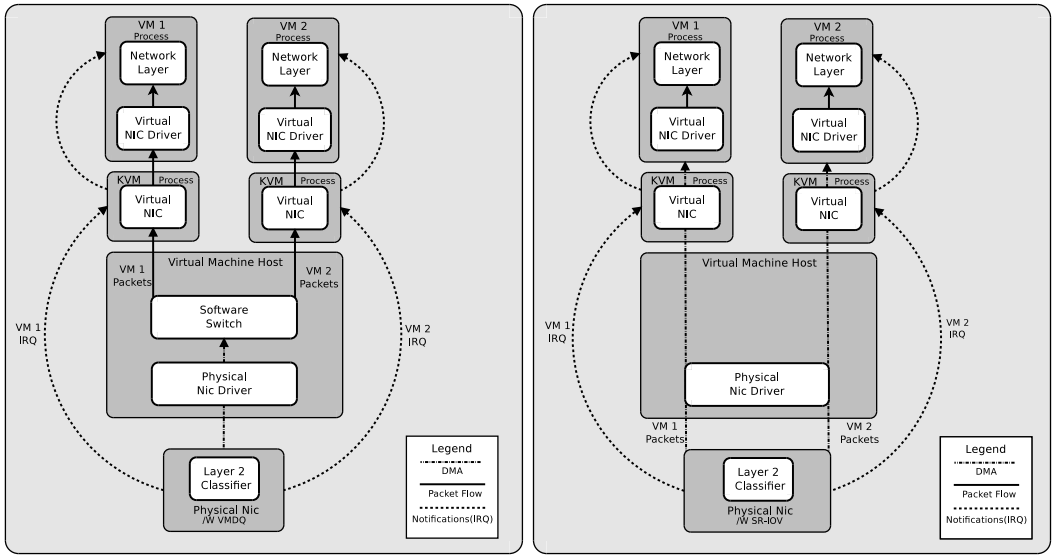


(a) Bare-Metal



(b) KVM

Figure 1.4: Packet Delivery Bare-Metal and KVM



(a) VMDQ (b) SR-IOV

Figure 1.5: VMDQ and SR-IOV

Figure 1.4 shows the different paths taken by a packet in the bare-metal system and KVM. In the bare-metal system, requests for access to the network stack and device drivers are often handled by the same kernel process. Because of this, there are far fewer context switches, fewer cycles consumed, and less cache used, since the same process is responsible for processing the packet and delivering it to the application layer. KVM however is in a strikingly different situation, in which a packet must traverse multiple processes in order to be serviced. This is because when the packet is received on the physical device, it is copied by the host machine's kernel into the memory space of the virtual machine; the virtual machine is then notified of the incoming packet and it is scheduled to run on the processor; the virtual machine delivers the packet to its network stack where it is finally delivered to the application layer. All these extra steps, when applied to each packet, contribute to the increase in CPU cycles, cache usage, context switches and interrupt requests we witnessed in the micro experiment.

1.5 Hardware-assisted Virtual Interfaces: Design and Performance

It is clear from our first micro-experiment that using software-based virtual interfaces is much more expensive than bare metal interfaces. Hardware manufactures have taken notice of this high overhead and have created devices which greatly help alleviate virtual NIC overhead. Two popular solutions are Virtual Machine Device Queues (VMDQ)[7] and Single

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

Root I/O Virtualization (SR-IOV)[8]. Both of them are referred to as VT-C by Intel, and both work by duplicating components of the NIC to give a VM more direct access to the hardware. Note however that SR-IOV has become an industry standard, and therefore different companies are free to package it in their network interface cards.

1.5.1 Virtual Machine Device Queues (VMDQ) and Single Root I/O Virtualization (SR-IOV)

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

VMDQ solves the above problem by only interrupting the VM that the packet was destined for. To do this, VMDQ contains multiple interrupts which can be assigned to different VMs. On receipt of a packet, a VMDQ enabled NIC uses a layer 2 classifier to determine which VM the packet is destined for based on either MAC address or VLAN tag. The NIC then uses direct memory access (DMA) to copy the packet into the host's memory, and sends an interrupt to the destination VM. The hypervisor then collects the packet, copies it into the VM memory space and notifies the VM.

Although VMDQ goes a long way to solve some of the overhead and scalability associated with virtual NICs the hypervisor must still copy the packet from the host's memory space into the VM space. To solve this, hardware manufactures have standardized SR-IOV, which, when applied to network cards removes the extra packet copy. To accomplish this, a SR-IOV enabled NIC can not only be configured to send an interrupt to the proper core but also DMAs the packet directly into VMs memory space. Using SR-IOV, virtual machines can not only save considerable overhead but also hit prior unattainable speeds such as 10 Gb/s [43]. Figure 1.5 shows how packets are delivered from the network to Virtual Machines using both VMDQ and SR-IOV.

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

	Bare-Metal	KVM SR-IOV	KVM VirtIO
Cycles	11.5M/Sec	31.1M/Sec	51.8M/Sec
LLC References	0.48M/Sec	2.1M/Sec	2.3M/Sec
Context Switches	189/Sec	476/Sec	452/Sec
IRQs	600/Sec	1200/Sec	2600/Sec

Table 1.3: Iperf receiving 10 Mb/s TCP traffic

	Bare-Metal	KVM SR-IOV	KVM VirtIO
Cycles	4.1M/Sec	18.8M/Sec	33.4M/Sec
LLC References	0.13M/Sec	0.72M/Sec	1.3M/Sec
Context Switches	19/Sec	239/Sec	307/Sec
IRQs	180/Sec	390/Sec	1700/Sec

Table 1.4: Iperf sending 10 Mb/s TCP traffic

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

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

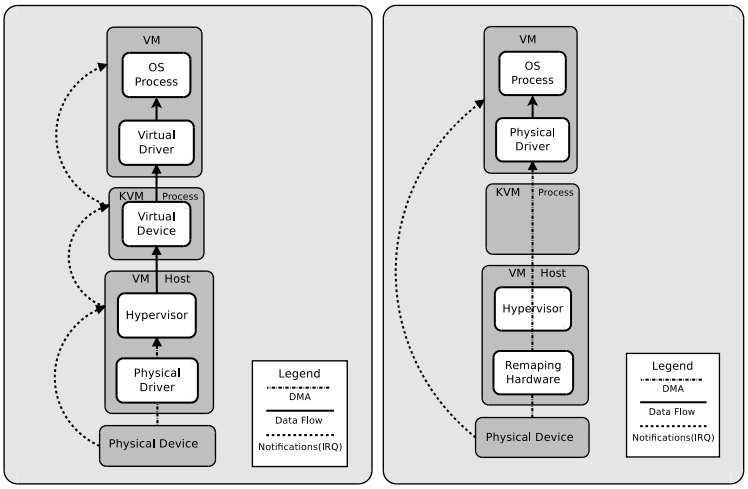
To show the performance overhead of these hardware-assisted virtual NICs we devised another micro experiment to compare SR-IOV, bare-metal and paravirtualized devices in terms of overhead. Once again we use KVM and compare its performance while using the paravirtualization drivers VirtIO and a SR-IOV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

Table 1.3 gives the results for receiving data and Table 1.4 the results for sending. As can be seen from both tables SR-IOV greatly reduces the overhead of the network traffic in nearly every metric when compared to VirtIO. For both sending and receiving the hardware assisted SR-IOV consumes approximately 40% less cycles than the paravirtualized driver VirtIO. However, when compared to the systems bare-metal performance SR-IOV still consumes several times more processor cycles. When compared to VirtIO the cache performance of SR-IOV is also improved since the packets are now directly copied into the virtual machine's memory space. In terms of processor context switches SR-IOV shows similar results to VirtIO. As described previously the hypervisor must still be switched on the processor to handle IRQs generated by the SR-IOV enabled NIC and this is likely why we do not see a great improvement in terms of the numbers of context switches. Finally, in terms of both sending and receiving SR-IOV greatly reduces the number of IRQs generated by the virtual machine. This is expected as now the interrupt can be directly handled by the virtual machines hypervisor.

In summary, we can see that the use of hardware devices can greatly decrease the amount of overhead experienced by a virtual network interface, however, there is still much room for improvement since in all our tested metrics SR-IOV still consumes several times the amount of resources when compared to a bare-metal network interface. Further, there has been a slow uptake of SR-IOV NICs in public cloud computing platforms. This delay in uptake is partially caused by security concerns around this new technology. Effectively, SR-IOV gives a public cloud user a direct hardware plug into the network of the data centre. Also, the public cloud provider may rely on some statistics collected at the software-based virtual NIC to generate accounting and billing information. Due to these security and logistical concerns, public cloud providers may be unwilling to replace a software-based NIC with a hardware-based SR-IOV NIC.

1.6 Graphics Processing Unit (GPU) Virtualization

A GPU consists of hundreds or even thousands of cores, allowing a vast amount of threads to run simultaneously to solve computationally intensive tasks. Due to the intrinsically parallel nature, a GPU demands much higher memory bandwidth than a CPU, and thus existing hardware designs around GPU are mostly throughput driven. For instance, the GPU's internal bandwidth is optimized by using a dedicated memory, the newest generation of which is GDDR5. Externally, GPU is interfaced with the motherboard by a PCI Express (PCI-E) expansion slot, which has a much higher bus throughput than other older PCI buses. Unfortunately, the data transfer between the main memory and GPU is still a bottleneck, and virtualization can dramatically degrade the memory transfer performance. Furthermore, using SIMD (Single Instruction, Multipule Data), a GPU is able to unleash its computing power on data-parallel computations. However, this makes it harder to share



(a) Shared Device (b) Pass-through Device

Figure 1.6: Shared vs Pass-through Device Simplified Architecture

a GPU among multiple VMs, each with their own distinct task to perform. The CPU, on the other hand, excels at task-parallel computations with its multiplecores, making virtualization easier with concurrent access to different VMs and their disparate tasks.

1.6.1 GPU Architecture and Pass-through

Recent hardware advances have enabled virtualization systems to perform a one-to-one mapping between a device and a virtual machine guest, allowing hardware devices that do not virtualize well to still be used by a VM, including GPU. Both Intel and AMD have created hardware extensions for such device *pass-through*, namely VT-D by Intel and AMD-Vi by AMD. They work by making the processor's *input/output memory management unit* (IOMMU) configurable, allowing the systems hypervisor to reconfigure the interrupts and *direct memory access* (DMA) channels of a physical device, to map them directly into one of the guests [1].

As illustrated in Figure 1.6a, data flows through DMA channels from the physical device into the memory space of the VM host. The hypervisor then forwards the data to a virtual device belonging to the guest VM. The virtual devices interacts with the driver residing in the VM to deliver the data to the guest's virtual memory space. Notifications are sent via interupts and follow a similar path. Figure 1.6b shows how a 1-1 device pass-through to a VM is achieved. As can be seen, the DMA channel can allow data to flow directly from the physical device to the VMs memory space. Also, interrupts can be directly mapped into the VM through the use of remapping hardware, which the hypervisor configures for the guest VM.

The advanced pass-through grants a single VM a one-to-one hardware mapping between itself and the GPU. These advances have allowed the cloud platforms to offer virtual machine instances with GPU capabilities. For example, Amazon EC2 has added a new instance class known as the `GPU Instances`, which have dedicated NVIDIA GPUs for graphics and general purpose GPU computing.

The performance implications and overhead of pass-through have been examined for various pass-through devices, e.g, network interface controllers [43][20][14]. There have also been recent studies on enabling multiple VMs to access CUDA-enabled GPUs [63][54], analyzing the performance of CUDA applications using a GPU pass-through device in Xen [71]. As well as GPU resource scheduling in cloud [73][75].

Although many cloud computing workloads do not require a GPU, cloud gaming servers require access to a rendering device to provide 3D graphics. As such VM and workload placements have been researched to ensure cloud gaming servers have access to adequate GPU resources. In 2011, Kim *et el* [38] proposed a novel architecture to support multiple-view cloud gaming servers, which share a single GPU. This architecture provides multi-focal points inside a shared cloud game, allowing multiple users to potential share a game world, which is rendered on a single GPU. In 2012, researchers at the University of Southern California performed an analysis of the performance of combined CPU/GPU servers for game cloud deployments [77]. The researches tested offloading different aspects of game processing to these cloud servers, while maintaining some local processing at the client side. They concluded that leaving some processing at the client side could lead to an increase in QoS of cloud gaming systems.

Pioneering research has also been done on GPU sharing and resource isolation for cloud computing servers [74, 51, 76]. These works show that with proper scheduling and allocation of resources we can maximize GPUs utilization, while maintaining high performance for the users sharing a single GPU. In 2013 it was shown that direct GPU assignment to a virtualized gaming instance can lead to frame rate degradation of over 50% in some gaming applications [60]. The researchers found that the GPU device pass-through severely diminished the data transfer rate between the main memory and the GPU. In 2015, the work was extended to more advanced platforms and found that although the memory transfer degradation still existed it no longer effected the frame rate of current generation games [59]. Hong, *et el* [25] performed a parallel work, where the researchers discovered that the frame rate issue present in virtualized clouds may be mitigated by using mediated pass-through, instead of direct assignment. Work has also been done on encoding for applications such as cloud gaming. For example, recent research has show decoding complexity and battery life of mobile devices can be greatly effected by encoding parameters set at the server side [40]. Further, Hemmati *et al.* [23][24], examined the player's experience of cloud gaming with selective object encoding methods.

In the next chapter we will cover security issues inherent to virtualized environments.

Chapter 2

Security Issues in Virtual Machine Based Environments

With any new technology there are downsides and tradeoffs; virtualization is no exception to this rule. With the ability to run multiple different VMs and even different operating systems at the same time comes the cost of considerable overhead. This overhead effects all components of the system; however it has been noted to be especially high in terms of network performance [49]. Given the inherent overhead of networking on virtualized systems, it is a surprise to find that very little research has been done to quantify the effect of networked Denial of Service (DoS) attacks on these systems.

In this chapter, we explore the performance of modern computer virtualization systems under a TCP based Denial Of Service (DoS) attacks. We devise a representative set of experiments to examine the performance of some typical virtualization techniques under TCP-based DoS attacks. By comparing the performance of virtualized and non-virtualized systems under a TCP based DoS attack we show that virtualized systems are much more susceptible to networked DoS attacks than their non-virtualized counter parts. Even with relatively light attacks, the file system and memory access performance of hypervisor-based virtualization degrades at a much higher rate than their non-virtualized counterparts. Further, while under a DoS attack we show a web server hosted in a virtualized system can serve 23% fewer request per second, while our bare-metal server hosted in the same hardware degrades by only 8%. These observations suggest that the state-of-the-art virtualization solutions need to be substantially revisited in this perspective. We further examine the root causes, with the goal of enhancing the robustness and security of these virtualization systems. With the data gained from our experiments, we show that with a few clever modifications hypervisor-based virtualization systems can be made much more resistant to networked DoS attacks. We implement our modifications into a real-world virtualization system and show we can mitigate the overhead of a DoS attack by up to 40%.

2.1 Virtualization Systems to Be evaluated

To thoroughly analyze and compare virtualization techniques under DoS attacks, we need to select representative samples of virtualization packages, to cover the typical and state-of-the-art solutions. As previously mentioned in chapter 1.2, all current virtualization solutions can be classified into three main categories, Paravirtualization (PVM), Hardware Virtualization(HVM) and container based virtualization.

In our experiments, we chose a representative of each class of virtualization. Namely, Xen, KVM and OpenVZ to be evaluated under DoS attacks. We believe this choice is representative for the following two reasons. First, they are all open-source with publicly available documents and with cross-platform implementations. We can run their packages on the same platform without changing OS or computer hardware. This makes a fair comparison possible and the results reproducible. Second, all of them have been widely used in real-world production environments for server consolidation and Cloud Computing. As mentioned previously, Xen has been used heavily to provide Cloud Computing functionality, for example in Amazon EC2; KVM has been used by Ubuntu Enterprise Cloud and Eucalyptus Cloud Service [46]; OpenVZ is a popular choice in offering Virtual Private Server(VPS) containers to the public.

There have been many performance analyses performed on different applications and scenarios in virtualized systems [30][18][31]. In 2007, researchers from the University of Michigan and HP conducted a performance evaluation comparing different virtualization techniques for use in server consolidation [49]. They compared Xen, a hypervisor-based paravirtualization technique, and OpenVZ, a container-based virtualization technique. The results showed that OpenVZ had better performance and lower overhead than Xen.

Soltesz *et al.* [65] compared Xen and Linux VServer in terms of performance and architectural design. Matthews *et al.* [45] tested HVM, PVM and Container Virtualization for performance isolation. They found that HVM has better performance isolation, followed closely by PVM, and that container-based solutions provide the least isolation.

Recently, Ostermann *et al.* [48] conducted a performance analysis on Amazon EC2 to determine its suitability for high performance scientific computing. They found that the use of virtualization can impose significant performance penalties on many scientific computing applications. The impact of virtualization on network performance in Amazon EC2 was evaluated in [68]. It showed that, due to virtualization, users often experience bandwidth and delay instability.

Despite these pioneer works on quantifying the overhead and performance of virtualization under various environments, to our knowledge, the performance of virtualization under networked DoS attacks remains largely unexplored.

2.2 Overview of Denial Of Service (DoS)

Denial of Service (DoS) attacks are attempts by a non-legitimate user to degrade or deny resources to legitimate users. There are several forms of DoS attacks. In this chapter, we will focus on networked DoS, the most common threat against modern IT infrastructure. In particular, we examine the TCP SYN flood attack against a target machine, which is one of the most common attacks on the Internet today and is notoriously difficult to filter out before it reaches the end system. As we will show later, however, that our findings are indeed general and not simply confined to TCP SYN flood. In this section, we first give a brief discussion on the TCP SYN flood attack and its potential threat to virtual machines.

2.2.1 TCP SYN Flood

The Transmission Control Protocol (TCP) is one of the foundations of the global Internet. TCP provides reliable in-order delivery of whatever data the users wish to send. When TCP was initially developed, the Internet remained a small private collection of computers and security issues inherent in the protocol were of little concern. As such some features of TCP can be exploited to perform DoS attacks.

The TCP SYN flood is one of the simplest and most common attacks seen on the Internet. This attack takes advantage of the amount of resources that have to be allocated by a server in order to perform a 3-way handshake. An attacker tries to overload a victim with so many connection requests that it will not be able to respond to legitimate requests. This is done through sending many TCP SYN packets to the victim. The victim allocates buffers for each new TCP connection and transmits a SYN-ACK in response to the connection request. The attacker has no intention of opening a connection, so it does not respond to the SYN-ACK [17]. Flooding based attacks can also exhaust other resources of the system such as CPU time.

2.2.2 TCP DoS Mitigation Strategy

Many defenses have been proposed to combat the TCP SYN flood. The simplest is to use a firewall to limit the number of TCP SYN packets allowed from a single source. However, many attacks use multiple hosts or employ address spoofing. The case where multiple hosts are involved in an attack is often called a Distributed Denial of Service attack (DDoS). More complex solutions have met with a better level of success and are usually deployed either in the network or on the end host. Network-based solutions include firewall proxies, which only forward the connection request after the client side ACK is received [17]. In particular Hop Count filtering, inspects the packet's TTL field and drops suspected spoofed packets. It has been reported that this technique can achieve up to 90% detection rate [69][34]. End point solutions include SYN cookies and SYN caches, both of which have been widely deployed. SYN caches work by allocating the minimum amount of data required when a SYN packet

arrives, only allocating full state when the Client's ACK arrives [16]. SYN cookies allocate no state at all until the client's ACK arrives. To do this, the connection's states are encoded into the TCP SYN-ACK packet's sequence number; on receipt of the ACK, the state can be recreated based on the ACK's header information [7].

Since virtual machines interact with the network through their virtual interfaces in much the same way that physical machines interact with the network, many of the considerations and defenses for DoS attacks mentioned above apply to virtualized systems. However, it is well known that current hypervisor-based virtualization can experience high overhead while using their I/O devices such as the network interface. Since DoS attacks attempt to exhaust resources on a targeted server, the stresses on the network interface would amplify the virtualization overhead and thus become even more effective at degrading the target. This will be demonstrated by our experimental results, even though such preventive strategies as SYN cookies and caches have been enabled in our experiments.

2.3 Experimental Architecture

To evaluate each virtualization technique, we created a small scale yet representative test network and system in our lab. We chose to create our custom test system instead of using rented resources from a cloud provider for several reasons. The first is that cloud providers such as Amazon EC2 have specific rules regulating security testing on their systems. To our knowledge all large cloud providers specifically list DoS testing as a non-permissible activity. Second, using our own custom system is the only way to ensure hardware resources remain constant across several different tests. Finally, no cloud provider provides its users with direct access to the Virtual Machine host, making some measurements impossible to perform. We now give a detailed description of the hardware and software used in our tests.

2.3.1 Physical Hardware and Operating System

We used a modern mid-range PC with an Intel Core 2 Q9500 quad core processor running at 2.83 Ghz. We enabled Intel VT-X in the bios as it is required for Hardware Virtual Machines (HVM) support. The PC was equipped with 4 GB of 1333 MHZ DDR-3 SDRAM and a 320 GB 7200 RPM hard drive with 16MB cache. The network interface is a 1000 Mb/s Broadcom Ethernet adapter attached to the PCI-E bus.

The host and the guests all used Debian Squeeze as their operating system. The kernel version remained constant at 2.6.35-5-amd64 across all tests. Since Xen and OpenVZ require special kernel patches, we used 2.6.35-5-Xen and 2.6.35-5-OpenVZ for those tests. In all tests, we use the amd64 version of the kernel and packages.

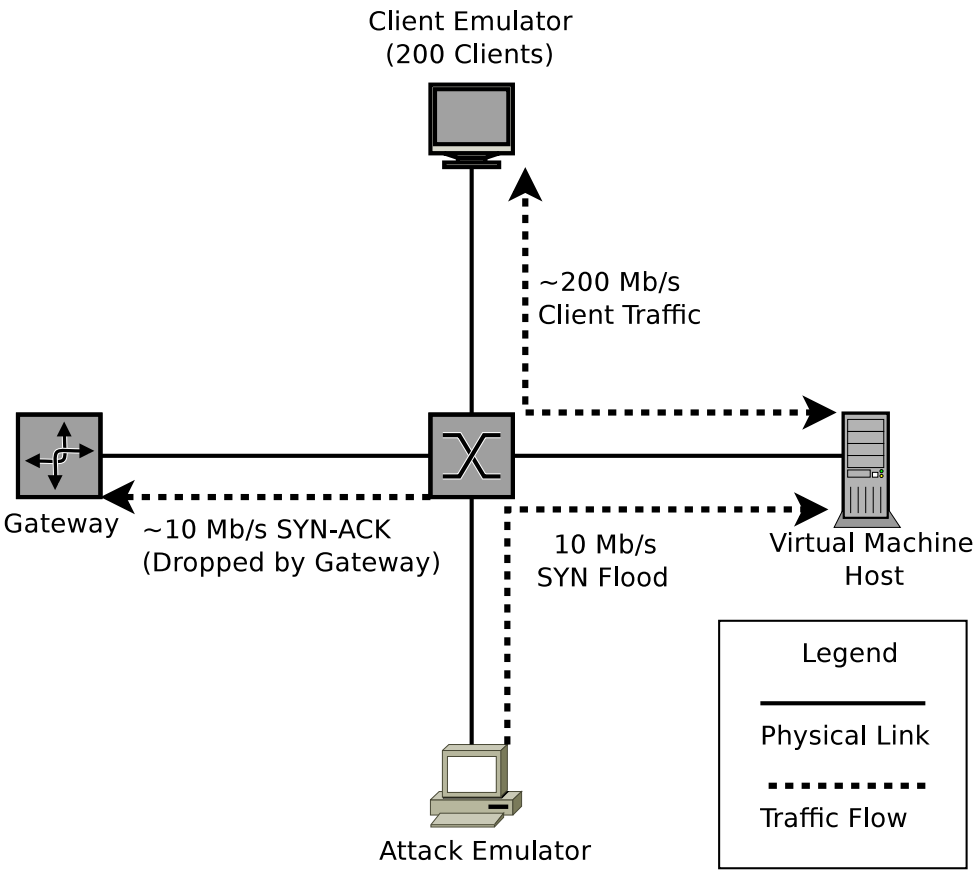


Figure 2.1: Network Setup and Traffic Flow

2.3.2 Network Setup

To emulate a DDoS attack against our servers, we employed the network configuration shown in the Figure 2.1. The figure also shows traffic flows for our benchmarks. All machines on the network are attached directly to a Linksys 1000Mb/s SOHO switch. The attack emulator has been configured to have a link rate of only 10Mb/s using the `ethtool` configuration tool. The client emulator used in our experiments was a dual core Pentium D PC, which created clients for our comprehensive benchmarking. The gateway is the default route for any host outside this directly connected subnet. When we simulate an attack, the gateway is configured to drop forwarded packets. Dropping these packets makes it appear as though they have been forwarded to an external network from the perspective of the virtual machine host. If the gateway were not present, many gratuitous ARP requests would have been created by the virtual machine host, as it searches for a route to deliver the packets to the external network. Our choice to place the Client Emulator and Attack Emulator inside the gateway may seem unintuitive. However, this is done for the following reason: To ensure the accuracy of our tests we must only allow the DoS attack to degrade the virtual machine host and not other network components. Also large networks which perform load-balancing

or traffic filtering, often have different routes for received and transmitted data, making it not uncommon to find similar network setups in practice.

2.3.3 Emulating a Denial Of Service Attack

The DDoS attack simulated for our experiments is the standard TCP SYN flood. Our choice of this attack was motivated by the fact that it is one of the most common DoS attacks seen on the Internet today. It is also notoriously hard to be filtered out from legitimate traffic. Yet most of our observations and conclusions are general, not simply confined to TCP SYN flood. In our experiments, we assume that a 100Mb/s distributed SYN flood is being performed on our network and we have successfully filtered out 90% of the attack. A total of 10Mb/s of attack TCP SYN traffic has bypassed the detection and reached the end host. We believe this is a reasonable setting as no existing solutions have been shown to effectively filter out all attack traffic without greatly effecting legitimate clients [69]. On the end host, we have enabled the SYN cookies defense and, by default, the Linux Kernels use the SYN Cache defense.

To generate the actual attack, we used the open source `hping3`[1] tool. The tool allows us to create arbitrary packets with which to flood a target host. We set hping3 to create TCP SYN packets and randomly selected the source addresses. We target an open port on the target machine. In the case of our synthetic benchmark, it is the SSH port 22 and in our comprehensive benchmarks, it is Apache2 running on port 80. The DoS traffic originates from our attack emulator, which can be seen in Figure 2.1.

2.3.4 Virtualization Setup

As explained earlier, we have chosen Xen, OpenVZ, and KVM in our experiments for their open-source nature and their extensive deployment in the real-world. We now describe the system setup of these virtualization solutions.

Xen System Setup

We installed the Xen 4.0 Paravirtualization Hypervisor on our test system following closely the Debian guide. To configure networking we created a bridged adapter and attached our primary interface and Xen's virtual interfaces to it. Xen virtual machines received an IP address from the DHCP running on our gateway. For disk interface, we used Xen's LVM features as we already had LVM running on our host machine. To install a base system into the image, we used the utility `xen-tools`, which automates the install procedure. We set the number of virtual CPUs (VCPU) to 4 and the amount of RAM to 2048 MB. The virtual machine host ran the 2.6.35-5-Xen kernel.

[1] `http://www.hping.org/hping3.html`

OpenVZ System Setup

We installed the OpenVZ container-based virtualization package from the Debian repository using its guide. We configured our container using the Debian Squeeze template. The container was given access to 2048 MB of main memory and complete access to the 4 CPU processing cores. Like the Xen setup, network access was provided to the container by bridging the container virtual Ethernet interface to our physical interface. The virtual machine host ran the 2.6.35-5-OpenVZ kernel.

KVM System Setup

KVM is relatively simple to install since it only requires a kernel module instead of a special patched kernel. We used KVM version 0.12.5 compiled from the official source repository. We manually installed an instance of Debian Squeeze from a mounted CD image into the KVM virtual machine. Once again the virtual machine was given access to all 4 processor cores as well as 2048 MB of memory. The disk interface was configured as a flat file on the physical host's file system. Networking was configured once again as a bridge between the virtual machine's interface and the system's physical NIC. To enable the best network performance, we configured KVM to use the VirtIO network drivers [56]. Debian kernel 2.6.35-5-amd64 was used in the virtual machine to stay consistent with the other tests.

Non-Virtualized "Vanilla" System Setup

Finally, as the baseline for comparison, we had a bare-metal "Vanilla" setup with no virtualization running, i.e., the system has direct access to the hardware. The same drivers, packages and kernel were used as in the previous setup. This configuration enabled us to calculate the minimal amount of performance degradation that our system can experience.

2.4 Benchmark Setup

We have chosen a broad set of synthetic benchmarks and a single comprehensive benchmark to evaluate the impact of DoS attacks on different components; specifically CPU, Network, Memory, and File System performance under normal and attack conditions.

2.4.1 CPU Benchmark

We chose the `SysBench`[2] CPU test to measure the CPU performance, which is well known and regularly used for gauging raw CPU performance.

It continues to calculate prime numbers until a threshold chosen by the user is reached and the results are presented as the total time to calculate the primes. We chose to find

[2] `https://launchpad.net/sysbench`

all primes less than 100,000 and assigned 4 threads, so that each of our 4 cores would be involved in the benchmark. We then recorded the total amount of time it took to find the primes in the first 100,000 integers.

2.4.2 Memory Benchmark

For memory benchmarking, we chose the SysBench memory bandwidth test. It allocates a segment of either global or thread local memory and performs read or write operations on it, outputting the total time taken as well as the memory bandwidth in MB/s. In our experiments, we assign a single thread to perform a 20GB read of main memory.

2.4.3 File System Benchmark

The file system performance was tested using the SysBench 'fileio' test, which creates a specified set of files on the disk and performs read and write operations on them. For our experiments, we created 2 GB worth of files and performed random read and write operations on the files. We assign 16 threads each attempting to perform random blocking read or write operations. The results are given as MB/s.

2.4.4 Network Benchmark

To test network performance we use `Iperf`[3], which attempts to find the maximum TCP or UDP bandwidth(in Mb/s) between two network hosts. In our experiments, the client emulator in our network was chosen as the server and TCP was used as the protocol. We run each test for the default time setting of 10 seconds.

2.4.5 Comprehensive Benchmark - Web Application

To further understand the overall system performance, we have devised a comprehensive benchmark based on a simple 2-tier Web Server and Database. We used the Debian repositories to install the `Apache`[4] 2.2 Web Server and the `MySQL`[5] Server 5.1. To create a web application representative of a real-world service, we installed the `RuBBoS`[6] bulletin board benchmark. We chose the PHP version of the RuBBoS and installed the necessary Apache extensions for PHP. We then installed the RuBBoS data into our MySQL database.

Although RuBBoS comes with its own client simulator, we used the Apache benchmark instead. The latter has been more commonly used for web server stress testing. Although the RuBBoS simulator can perform tests specific to RuBBoS, we only require maximum request rate, which are more straight forward to extract with the Apache Benchmark.

[3] `https://iperf.fr/`
[4] `https://httpd.apache.org/`
[5] `https://www.mysql.com/`
[6] `http://jmob.ow2.org/rubbos.html`

	TCP DoS		UDP Flood	
System	10Mb/s	100Mb/s	10Mb/s	100Mb/s
KVM	~102%	~205%	~64%	~272%
Xen	~98%	~187%	~46%	~145%
OpenVZ	~8%	~100%	~1%	~1%
Vanilla	~6%	~90%	~1%	~1%

Table 2.1: CPU Usage While Under Attack – System Idle

We ran the Apache Benchmark against the RuBBoS website in each of the test setups. We simulated 200 clients requesting the latest forum topics page. By using this page, the web server must perform a single SQL query and render the PHP page for each user request. We then used the Apache benchmark to calculate how long it takes to service 500,000 requests. Figure 2.1 shows the network configuration and the traffic flows during this experiment.

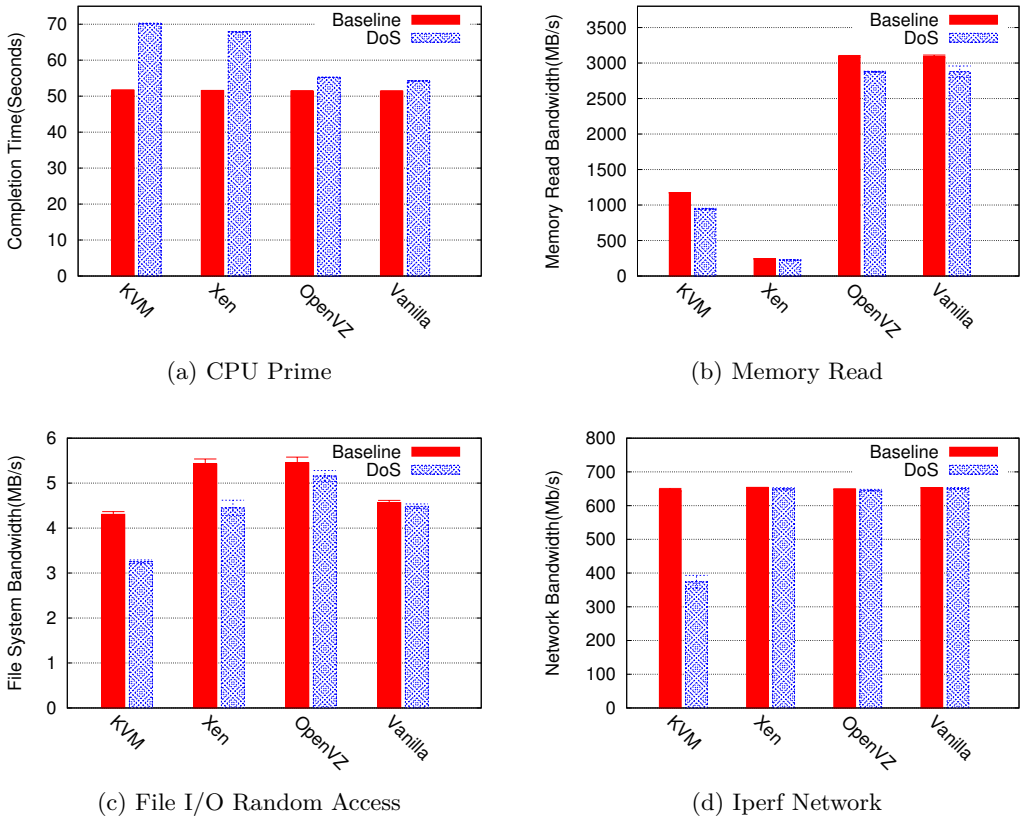
2.5 Experimental Results

2.5.1 CPU Usage During DoS

We first measure the impact of DoS traffic on CPU usage while the system is idle. To provide a target for the TCP DoS, we ran a Secure Shell(SSH) server and configured it to listen to port 22. For KVM, OpenVZ and Vanilla, the measurements were performed using the Linux `top` command. Since Xen is not compatible with the standard Linux top command, we used the xentop command to measure the CPU usage. We ran each system under both 10 Mb/s and 100 Mb/s TCP SYN floods. We also included the CPU usage from a 10 Mb/s and 100 Mb/s UDP flood. The results are given in Table 2.1.

Under a 10Mb/s TCP DoS, the hypervisors in both KVM and Xen consume the CPU time of an entire core simply delivering SYN packets to the VM and returning SYN-ACKs to the network. OpenVZ and Vanilla, on the other hand, use only between 6-8% of CPU time on a core to service the same attack. If we increase the attack rate to 100Mb/s, all systems increase their CPU usage; however both Xen and KVM consume nearly half of the systems total CPU resources simply processing the attack traffic. As we increase the attack traffic rate, the corresponding increase in CPU usage indicates that the systems will continue to degrade as it is exposed to higher attack rates.

Though our focus is on TCP SYN attack, we have also devised a UDP flood to determine if TCP was the culprit for the massive CPU usage experienced by the virtual machines. To create the UDP flood, we once again used hping3 with 10Mb/s and 100Mb/s of UDP traffic targeted the system. To further simplify the test we did not randomize the source address and the packets contained no data. By not randomizing the source address we can determine if it is the cause of the CPU overhead. As can be seen from Table 2.1, the



(a) CPU Prime

(b) Memory Read

(c) File I/O Random Access

(d) Iperf Network

Figure 2.2: Synthetic CPU, Memory, File System and Iperf Benchmark Result

high CPU usage is present for both KVM and Xen in the UDP case as well, suggesting that this happens whenever the hypervisor experiences a data stream that contains small packets at a high rate. We also tested ICMP and plain IP packets and found that any small packets sent at a high rate reaching the end system leads to a similar phenomenon. Other researchers have taken notice of this overhead during processing packets and have managed to reduce the CPU usage by 29% [32]. Even with this reduction, however, DoS attacks against hypervisor-based virtual machines still generate substantially more CPU usage than their counterparts.

Next, we benchmark our systems under a 10Mb/s TCP DoS to quantify the performance degradation experienced by the systems.

2.5.2 Synthetic Benchmarks Results

For each test, we ran each benchmark 4 times and calculated the mean. To ensure our results are statistically significant, we also calculated the standard deviation for each measurement and display it as error bars on our graphs. To quantify performance degradation, we calculate percentage change from baseline to attack conditions.

Result CPU Benchmark

The SysBench Prime benchmark is given in Figure 2.2a and a lower completion time is better in this case. We can see that in the non DoS case all virtualization techniques perform within 5% of each other. This is not surprising, since VM CPU scheduling is quite similar to OS process scheduling.

However, we can see from the CPU benchmark that even a small 10Mb/s DoS has a significant effect on the CPU performance of both KVM and Xen. KVM suffers from a massive 35.8% increase in completion time; Xen also suffers a large increase at 31.6%. For both Xen and KVM this increase is due to the amount of time the hypervisor spends on the CPU servicing the attack packets. OpenVZ and Vanilla fared much better, both with a small but still measurable amount of performance degradation of 7.5% and 5.5%, respectively.

Result Memory Benchmark

The memory benchmark results shown in Figure 2.2b are intriguing, as there is a wide variation in the base line performance. In this particular benchmark, Xen fared by far the worst, being over 12x slower than our Vanilla system setup. KVM fared much better than Xen but still only managed less than half of the memory read speed of Vanilla or OpenVZ.

Under the TCP DoS, all setups showed a measurable slowdown in performance, with Vanilla, OpenVZ and Xen having all approximately 7% performance degradation. KVM, on the other hand, experienced a slowdown of 19.4%. In KVM, the hypervisor must map a memory access request from the guest's memory address to the corresponding physical address on the host machine. We conjecture that the hypervisor is busy servicing I/O request created by the DoS packets. With this, the memory requests must wait longer to be mapped to the correct physical address. This delay manifests itself in the large performance degradation experienced by KVM in this test.

Results I/O benchmark

The synthetic benchmark results for the SysBench I/O test are given in Figure 2.2c. Although there is a significant difference in base line performances in this test, it is hard to make a direct comparison, due to the nature of disk benchmarking. For example, Xen and OpenVZ showed significantly faster performance than the others tested. However, the physical location on the disk where the 2 GB of files are allocated can make a large difference in bandwidth and seek time. For this reasons, we will refrain from comparing system's baseline performance on this benchmark and instead focus on the performance loss during a DoS.

Under the DoS conditions, OpenVZ and Vanilla degradation was within the deviation, indicating very little degradation. On the other hand, both KVM and Xen lose considerable

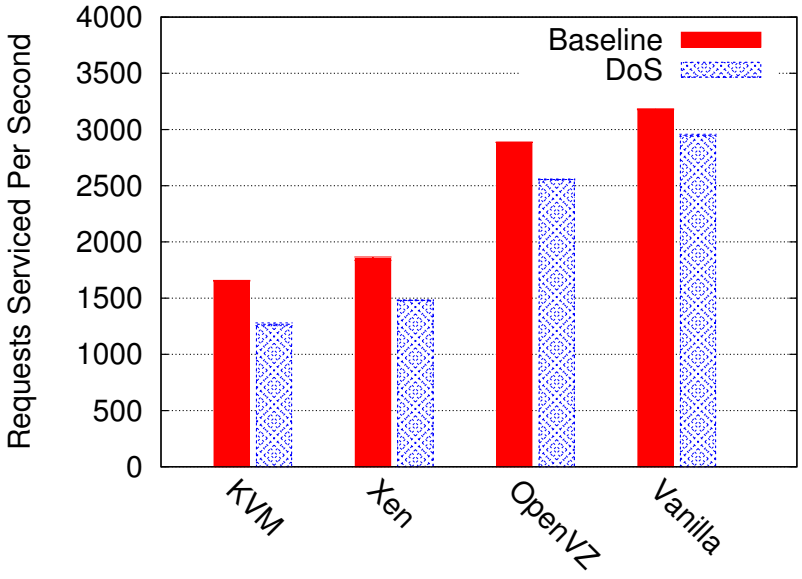


Figure 2.3: Web Application Benchmark Result

performance. Xen suffered from 18.0% lower random access to its file system. KVM lost the most performance, which is 24.5%. We believe that Xen and KVM's performance loss is likely due to the hypervisor delaying disk access and instead favoring to deliver attack packets to the virtual machine.

Result Network Benchmark

The Iperf results are given in Figure 2.2d. The Iperf benchmark measures maximum transfer rate between two hosts. The maximum transfer rate base line is almost identical for all systems.

Under TCP DoS attack, the IPerf results showed no significant drop in the throughput of Xen, OpenVZ or Vanilla systems. KVM however loses a massive 42.2% of performance for a total of 272 Mb/s less throughput. This is somewhat surprising considering the attack only creates 10Mb/s of TCP SYN packets, which can stimulate the KVM VM to produce at most 10Mb/s of SYN-ACK packets. This means KVM is experiencing a total of 20Mb/s extra traffic over the baseline. The loss of bandwidth however, is several times this extra traffic; in section IX we will show this phenomena is related to a high number of interrupt requests.

2.5.3 Comprehensive Benchmark Results - Web Application

From the previous synthetic benchmarks, it is clear that hypervisor-based virtualization techniques are more susceptible to performance degradation under DoS attacks. Although synthetic benchmarks are excellent at pinpointing performance bottlenecks, we further ex-

amine a more complex scenario to gauge the effect these attacks have on a real-world web system.

To perform our benchmark we first find the maximum throughput of each system. We express the maximum throughput as maximum number of requests serviced per second, which can be seen in Figure 2.3. OpenVZ suffers a measurable 9.3% decrease in throughput when compared to Vanilla. However, both KVM and Xen have significantly lower performance in their base line than both OpenVZ and Vanilla. Xen can service 41.2% fewer requests per second. KVM is the slowest, servicing 48.0% fewer request per second than our non-virtualized Vanilla system.

As expected, under DoS conditions, all systems experienced measurable performance degradation. Vanilla was the least susceptible falling by 7.2%, followed by OpenVZ at 11.4%. Xen suffered a 20.0% performance degradation and KVM lost 23.4%. When the systems are under attack, KVM and Xen provided nearly 50% less throughput than the Vanilla host provided while using the same amount of system resources.

2.6 A closer look: Causes of Degradation

In order to better understand the causes of the performance degradation that we have found, we take a closer look at these hypervisor-based virtualization systems. To accomplish this, we use hardware performance counters to measure interrupt requests (IRQs), processor cache activity, and CPU context switch events. Our goal is to compare the frequency of these events on virtualized and non-virtualized systems under denial of service attacks. With this information, we made modifications to the virtualization software in order to improve performance of the virtualized system under a DoS attack.

2.6.1 KVM Profiling

KVM was chosen as our representative hypervisor-based virtualization system in these tests, for several reasons. The first is that KVM runs on the unmodified Linux kernel, and is compatible with the standard performance profiling tools; Xen on the other hand, has a specialized kernel and therefore may have compatibility problems with these tools. Also, because KVM does not require a modified kernel, it is easier to modify the hypervisor and drivers. Second, KVM supports the VirtIO drivers, a para-virtualization package intended to be standard across various virtualization techniques, including Xen. By using VirtIO, any driver changes that are required in order to enhance performance will be easier to adapt to different virtualization systems.

To collect the system level information, we used `perf`[7], a performance analysis tool for Linux. It uses built-in Linux kernel support and hardware performance counters to measure

[7] `https://perf.wiki.kernel.org`

	Vanilla	KVM
LLC References	206,835,021	1,292,316,729
LLC Misses	11,486,718	36,060,083
Context Switches	956	708,335
IRQs/Second	8,500	37,000

Table 2.2: Performance Metrics Under DoS – System Idle

system-wide or process-level events. To collect the data, we disabled all non-operating system processes on our test system, effectively creating an idle state. The systems operated under the same specifications previously described. We then configured perf to track system wide events on all cores and collected our data during a 10Mb/s random source SYN flood DoS attack. To get an accurate reading we sampled the system performance 5 times, for 10 seconds each time and calculated the mean and standard deviation for each of our metrics. We then used the `perf top` command to obtain the current system statistics, in particular, IRQs per second.

As can be seen in Table 2.2, the samples show considerable difference in the metrics between the Vanilla system and KVM. In terms of LLC(last level cache) performance, KVM while under attack is considerably less efficient with the use of its LLC. KVM references its cache over 6 times more than the Vanilla system. In terms of last level cache misses, KVM is over 3 times worse. The number of cache misses and cache references is not surprising considering how much extra memory copying is required to deliver a packet from the network interface to its final destination in the KVM virtual machine.

Context switches also provided some interesting insights into the differences between virtualized and non virtualized systems. The Vanilla system, during our 10 seconds of measurements, had 956 context switches only, compared to a staggering 708,335 with KVM, an increase of over 700 times. This massive difference can be explained by how these two systems handle network packets. In the Vanilla system, services to the network stack and the driver domain are often handled by the same kernel process. Because of this, there are much fewer context switches as the same process is responsible for both actions. KVM however is in a strikingly different situation, in which a packet and its response must traverse multiple processes in order to be serviced. This is because when the packet is received on the physical device, it is copied by the host machine's kernel into the memory space of the virtual machine; the virtual machine is then notified of the incoming packet and it is scheduled to run on the processor; the virtual machine delivers the packet to its network stack where a response SYN-ACK is generated and sent on its virtual interface; the host kernel's process then collects the packet from the virtual machine and delivers it to the network. All these extra steps, when applied to each packet, contribute to the sheer number of context switches we observed. Figure 2.4 illustrates the path a received packet must traverse and the figure also shows a performance modifications we will discuss in the next section.

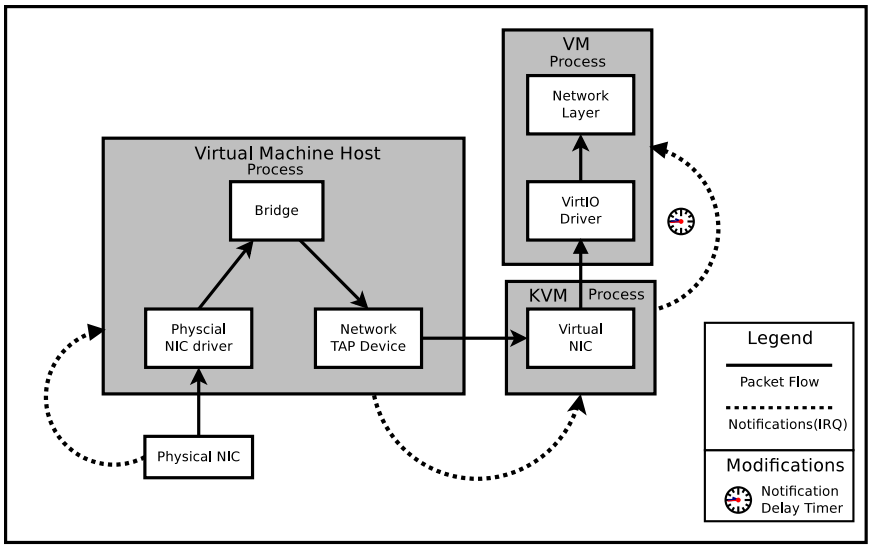


Figure 2.4: KVM Network Architecture and Modifications DoS

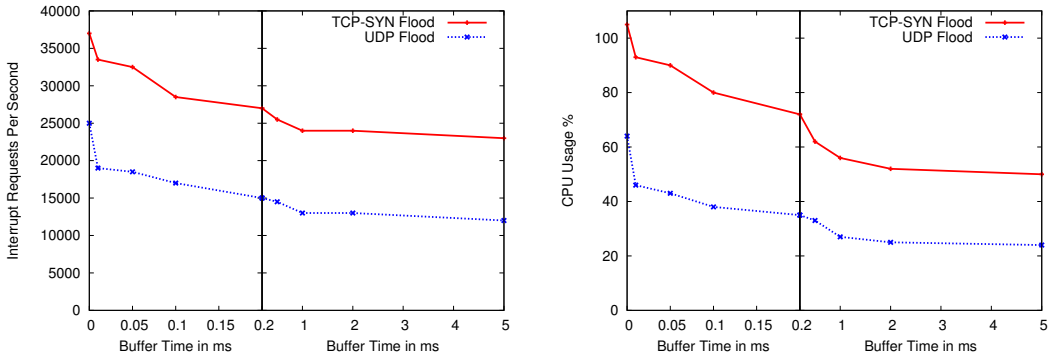
The final metric we looked at was interrupt requests. When under attack the Vanilla system generated approximately 8,500 interrupt requests per second. The KVM hypervisor generated over 4 times more requests, at approximately 37,000 interrupt requests per second. The interrupt requests are used by KVM to notify the virtual machine when it has new packets and also to notify the kernel when there are new packets to be sent. The TCP SYN flood results in many small packets being sent and received by the virtual machine, which results in the large number of requests we observed.

2.7 Reducing DoS Overhead KVM

Based on what we learned from our profiling of KVM, we decided to modify the VirtIO drivers in KVM to reduce the number of interrupt requests produced during a DoS attack, with the goal of improving the overall CPU performance of our virtualized system. We focused on interrupt requests specifically because they are a clear cause of context switches and ultimately affect cache efficiency as well.

2.7.1 Modifying KVM and VirtIO

As mentioned above, in KVM an interrupt request is generated whenever packets need to be sent from the host to the virtual machine, or from the virtual machine to the host. The KVM VirtIO drivers contain an option to bundle packets with a single interrupt for sending, but receiving lacks this feature. In our efforts to reduce the number of interrupts, we modified the VirtIO drivers to create a mechanism that bundles together several received packets with a single interrupt. To accomplish this, we examined the VirtIO driver and modified how KVM notifies the virtual machine that there is a packet to be received. When



(a) Reduction in IRQs per second

(b) Reduction in CPU

Figure 2.5: Reduction in Interrupts and CPU Overhead

the packet rate is high, instead of immediately notifying the virtual machine that it has received a packet, the KVM hypervisor sets a timer. When the timer expires, the virtual machine is notified of the initial packet, and every other packet that has arrived during the length of the timer. Figure 2.4 shows the KVM architecture and our modification. To make our modifications adaptive, they only become active when there is a high packet rate. To detect the packet rate, we keep a simple counter of the number of received packets and check it every second to see if the VM passes a certain threshold of packets per second; if the threshold is passed than the modifications are enabled; if the threshold is not reached we revert to the original VirtIO drivers. We applied the same technique to the sending queue as well. For our tests, we assume that a "high" packet rate threshold of 10,000 packets per second; however this is adjustable based on the application. Next, we experiment with the length of the timer, or how long to buffer the packets before generating an interrupt request.

2.7.2 Modification results

Using the same network and system specifications described in our previous experiments, we tested the effect that increasing the buffer time of our modified VirtIO driver had on interrupt requests per second and CPU usage. We tested candidate buffer times from 0 milliseconds up to 5 milliseconds. For each candidate buffer time, we measured the IRQs per second and CPU usage of our system while under a 10Mb/s TCP random source SYN flood and a 10Mb/s UDP flood. The results for IRQs per second are shown in Figure 2.5a, and the results for CPU usage are shown in Figure 2.5b. As can be seen, when we increase the buffer time, we see a decrease in both interrupts and CPU usage. This confirms that interrupts are closely related to CPU usage in KVM under attack conditions, and that decreasing interrupts will meet our goal of increasing CPU performance. UDP shows a very similar result, once again confirming that high CPU usage is tied to high packet rate and

	KVM	Mod-KVM
LLC References	1,292,316,729	684,267,176
LLC Misses	36,060,083	27,684,718
Context Switches	708,335	422,471
IRQs/Second	37,000	25,500

Table 2.3: Performance Metrics Under DoS – VirtIO Modifications

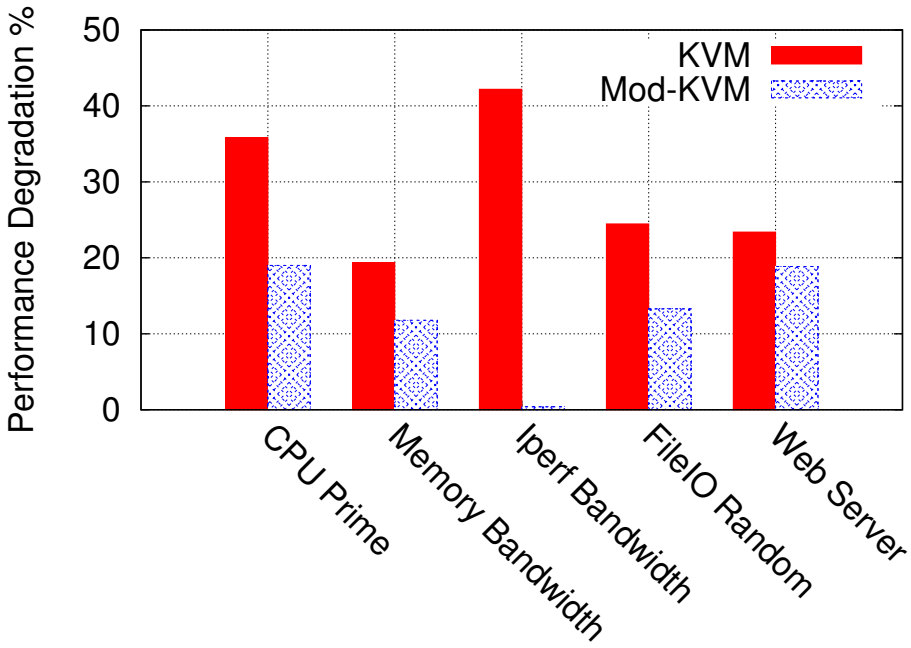


Figure 2.6: Improvement In Performance with 0.5 ms Buffer Time

small packet size. From our results it is clear that any amount of buffering time improves CPU performance, but that the improvement levels off after a certain point. We chose a buffer time of 0.5 ms for the rest of our experiments, which is a reasonable amount of delay and results in considerably lower CPU overhead. It is important to note that the delay choice affects both sending and receiving; therefore the round trip time of packet and its response can be delayed by at most 1 ms based on our choice.

Taking our modified VirtIO driver with the buffer time at 0.5 ms, we re-ran the profiling tests and compared this modified KVM to the original KVM. As Table 2.3 shows our modifications considerably reduced all of our metrics. There was a 47% decrease in cache references, and a 23% decrease in cache misses. This decrease occurs because the virtual machine is now using its cache more efficiently by processing more packets sequentially. Our modifications also reduced the number of context switches over 40% compared to the unmodified KVM, again because more packets are being processed at once.

We also re-ran the benchmarks from our previous experiment on our modified KVM virtual machine, again with the buffer time set to 0.5 ms. In Figure 2.6, we compare the performance degradation experienced by both KVM and our modified KVM under a DoS attack. Performance on all benchmarks was significantly improved by our modifications. In the CPU benchmark, our modifications reduces performance degradation from 35.8% to 19.0%; memory bandwidth no longer degrades by 19.4%, but instead by 11.8%. The modified drivers had the biggest impact on network bandwidth causing performance degradation to drop from 42.2% to less than 1%, effectively alleviating the problem; the file system benchmark improved from a degradation of 24.5% to 13.3%.

The modified KVM appeared to have the least improvement on the web server benchmark, improving from 23.4% degradation to 18.8%. However, a surprising result is found when we compare the actual number of connections per second: our modified KVM has a pre-attack baseline of 1,926 connections per second, whereas the original KVM has only 1,654 connections per second. This represents a considerable increase of 16.4% more connections per second. This is because our adaptive modifications to the VirtIO driver are activated even under the baseline conditions, meaning that combined traffic of the 200 simulated clients is greater than the threshold (10,000 packets per second). Therefore, while performance degradation only decreases by a small margin with our modified KVM, the absolute number of requests per second is much higher than for the original KVM, in both the baseline and attack situations. With this improvement, KVM is now faster in this test than even Xen was in terms of both baseline and attack conditions. These surprising improvements to the baseline indicate that our modifications may have applications to situations beyond DoS attacks.

2.8 Discussion

Despite the fact that modern virtualization has many attractive features, it is clear from our experiments that it is more vulnerable under TCP SYN DoS attack. In particular, although it has been known that virtualization entails higher overhead than traditional systems, our results suggest that even a light DoS attack can aggravate this overhead considerably. The combination of virtualization overhead and performance degradation in a DoS attack can lead to a 50% decrease in Web Server performance when compared to the non-virtualized Vanilla system using the same amount of resources. The performance implications of larger DoS attacks have yet to be quantified. We have preformed some preliminary experiments and it appears that, at higher DoS rates, hypervisor-based VMs may actually become unresponsive to the network.

Although defenses such as SYN-cookies and SYN-caches have been very effective at fortifying non-virtualized systems, we showed through our experiments that these measures do not provide adequate protection for VMs. It is also important to note that even though

the deployment of anti IP address spoofing technology such as ingress filtering (BCP 38) has reached the majority of autonomous systems, it is still estimated that up to 26.6% of autonomous systems allow spoofing[8]; this gap in deployment means this form of attack continues to be a real threat to VMs. SYN-proxies are another technology that may provide an effective mitigation strategy for VMs, however due to the effectiveness of endpoint defenses for non-virtualized systems there are currently many networks on which this technology is not deployed.

As shown in Table 2.1, any high packet rate with low packet size traffic pattern can cause performance issues for KVM and Xen. It remains to be discovered what effects similar non-DoS traffic have on real-world clouds such as the Xen-based Amazon EC2. Our initial trials using UDP Voice Over IP (VOIP) traffic indicate degradation will likely occur. As noted previously, our modifications to KVM actually greatly improved the baseline performance of KVM in our comprehensive benchmark. It is likely a wide range of applications with high packet rates could benefit as well: examples include VOIP and DNS.

Finally, it should be noted that in the global Internet, a 10 Mb/s SYN flood is considered rather small; however both the KVM and Xen hypervisors used the CPU time of an entire core, simply servicing that level of attack. When compared to OpenVZ and Vanilla, which used only 6% of a single core to serve the same attack, it becomes clear that hypervisor-based virtualization is significantly more expensive to use on systems exposed to DoS traffic.

[8]October 2015 estimate from the MIT Spoofer Project `http://spoofer.csail.mit.edu/`

Chapter 3

Energy Consumption Virtualized Clouds Analysis and Improvements

There have been significant studies on the performance and optimization of virtual machines (VMs), including their power consumption in performing different types of tasks. Research has been done on reducing power consumption of data-centers through the placing of energy consuming jobs/VMs in strategic cooling locations, energy reduction through job and VM consolidation, and energy aware scheduling [4] [3] [5] [66] [6] [44] [8]. Other pioneering works have explored solutions to meter or cap the energy consumption of virtual machines running in a cloud environment [37] [39]. Work has also been done on power consumed during a VM migration [27]. Further, the energy consumption implications of VMs used in server consolidation has been discussed in [35].

The precise power consumption of these virtualized systems while processing typical network transactions however has seldom been examined. Physical network interface cards are not known as power hungry, except for wireless communications. Yet the virtualization of the network module can be considerably more complex than others in a machine, given that it is largely a standalone unit with its own control and storage units, and with both sending and receiving functions across multiple layers. It is known that the network module can be a severe performance bottleneck in VMs [49]. It remains unclear what the precise implications of virtualized networking overhead will be on total system power consumption.

In this chapter, we present an empirical study on the power consumption of typical computer virtualization packages, including KVM, Xen and OpenVZ, while the systems are performing network tasks. We find that both Hardware Virtualization and Paravirtualization systems add a considerable amount of energy overhead to networking tasks. Both TCP sending and receiving can be noticeably affected, and a busy virtualized web-server may consume up to 40% more energy than its non-virtualized counterparts. We have conducted detailed profiling to analyze the workflow for packet delivery in virtualized machines, which reveals that a VM can take nearly 5 times more cycles to deliver a packet than a

bare-metal machine, and is also much less efficient on the system's hardware caches. The existence of hypervisors in VMs can dramatically increase interrupts and memory accesses, which in turn lead to significantly more power consumption for network transactions than a bare-metal machine does.

Our analysis further suggests that, without fundamental changes to the hypervisor-based VM architecture, the use of adaptive packet buffering potentially reduces the extra interrupts and memory copies. The practicality of our modifications has been validated through driver-level implementation as well as experiments in realworld systems. The results demonstrated that, with the adaptive buffering, we are able to improve the energy consumption of a busy web server by 16% without noticeable loss of its network performance.

3.1 Power Consumption Measurement: Platform and Virtualization Setup

In this section, we present the measurement configuration for the power consumption of virtual machines, including the hardware setup, the virtualization setup, and the measurement devices used.

3.1.1 Measurement Platform

For our test system, we use a modern midrange server with a Intel core i5 2400 3.1 GHz quad core CPU, 8 GB 1333 MHz DDR3 ram, a 500 GB 7200 RPM hard drive and a 1000 Mb/s Broadcom *Network Interface Card* (NIC) attached to the PCI-E bus. The choice of the Intel i5 server is motivated by the factor that the Intel's x86 architecture has long become dominating in the CPU market, which is also the basis for virtual machine implementation in such major cloud service providers as Amazon. The core i5 CPU is known to have low power consumption with well-designed state-of-the-art power management. In particular, when Intel introduced the Sandy Bridge line of processors, they also introduced the *Running Average Power Limit hardware counters* (RAPL) [29]. These highly accurate and versatile counters allow a user to configure the system to record its CPU power consumption. In all our experiments, we measure the internal power consumption using the RAPL counters while the CPU is running our network-based tasks.

Unfortunately, the RAPL counters do not measure the overall system power consumption. For example, in order to measure the power consumption of the modules other than the CPU, for example, cooling fans, hard drives, NICs, we have to use other tools. To determine the overall system's power consumption, referred to as *wall power*, we have wired a digital multi-meter (Mastech MAS-345) into the AC input power line of our system. We read the data from our meter over a PC-Link installed in the meter and collect samples every second throughout our experiments.

Finally, since our focus is on network transactions, we also configure a second system to work as a client emulator, which is a 2.8 GHz Intel Core2 Quad system, with 4 GB DDR3 ram, and a 1000 Mb/s network connection. We connect our test system and client emulator to each other through a 1000 Mb/s Linksys SD2005 SOHO switch.

3.1.2 Virtualization Setup

As explained earlier, we have chosen Xen, OpenVZ, and KVM in our experiments for their open-source nature and their extensive deployment in the real-world. We now briefly describe the system setup of these virtualization solutions.

Xen System Setup

We installed the Xen 4.1 Paravirtualization Hypervisor on our test system. To configure networking, we created a bridged adapter and attached our primary interface and Xen's virtual interfaces to it. The Xen virtual machine received an IP address from the DHCP server running on our gateway. Similar network configuration has also been used for the OpenVZ and KVM systems. For disk interface, we used the Xen's flat file feature. We set the number of virtual CPUs (VCPU) to 4 and the amount of RAM to 6144 MB. The virtual machine and physical host ran the 3.2.41-2-Xen kernel.

OpenVZ System Setup

We installed the OpenVZ container-based virtualization package from the official OpenVZ repository, following the latest guidelines. We configured our container using the Debian Squeeze template. The container was given access to 6144 MB of main memory and full access to the 4 CPU processing cores. The virtual machine host ran the ovzkernel-2.6.18 kernel, which is the latest patched kernel supported by Debian.

KVM System Setup

We compiled KVM version 1.2.0 from the official source repository. Once again the virtual machine was given full access to all the 4 processor cores as well as the 6144 MB of memory. The disk interface was configured as a flat file on the physical host's file system. The virtual machine and physical host ran the 3.2.41-2-amd64 kernel. To enable the best network performance, we configured KVM to use the VirtIO network drivers [56].

Non-Virtualized Bare-metal System Setup

Finally, as the baseline for comparison, we also had a bare-metal setup with no virtualization, i.e., the system has direct access to the hardware. The same drivers, packages and kernel were used as in the previous setup. This configuration enabled us to calculate the

minimal amount of energy required to run our benchmarks. Debian kernel 3.2.41-2-amd64 was used in this test.

3.2 Measurement Results and Analysis

To determine the power consumption profiles of the virtualized systems, we have performed a number of simple and complex benchmarks, and measured the corresponding CPU power consumption using the internal RAPL counter and the wall power of the systems using the AC power meter. We start from the following two benchmarks for network-oriented transactions:

Iperf Sending and Receiving: Before moving on to more complex experiments, we first used Iperf to determine the energy consumption of our systems while sending and receiving high bandwidth TCP streams. To this end, we configured our test system to communicate with our client emulator, sending and receiving at their maximum rates for a duration of 60 seconds.

Apache2 Many Client HTTP Download: In this experiment, we emulated the workload experienced by a busy web server that serves a number of clients. We configured our client emulator to open 100 HTTP connections, each downloading a 20 MB file, and then recorded the energy consumption experienced by each system.

To obtain the base-line power consumption of our systems, we also measured the power consumption when the system is in an 'idle' state. To this end, we killed all non-essential processes on the system. We measured each system for 60 seconds, taking samples of the power consumption (Watts consumed) every second.

To mitigate randomness, we run each experiment three time and calculate the average and standard deviation. We graph the results and display the standard deviation as the error bars on the graph. For experiments that have a deterministic running time we give the results in Watts (joules per second); for those with a non-deterministic running time, we give the results in total joules required to complete the benchmark. In each of our following experiments a single VM is running on the physical machine. Although the use of a single Virtual Machine is uncommon in practice, this configuration allows us to precisely measure the energy consumption of the virtualization system while running our test application. If multiple VMs were running on our host at the same time the energy consumption would increase as the VMs contended for the hosts resources.

3.2.1 System Idle Power Consumption

Figure 3.1 shows the baseline power consumption of our systems when idle. The Bare Metal, OpenVZ, and KVM systems all show similar results to one another for both the wall power (the external or whole system power consumption), and RAPL (the internal or CPU power consumption). These three systems all have an average power consumption of about 30

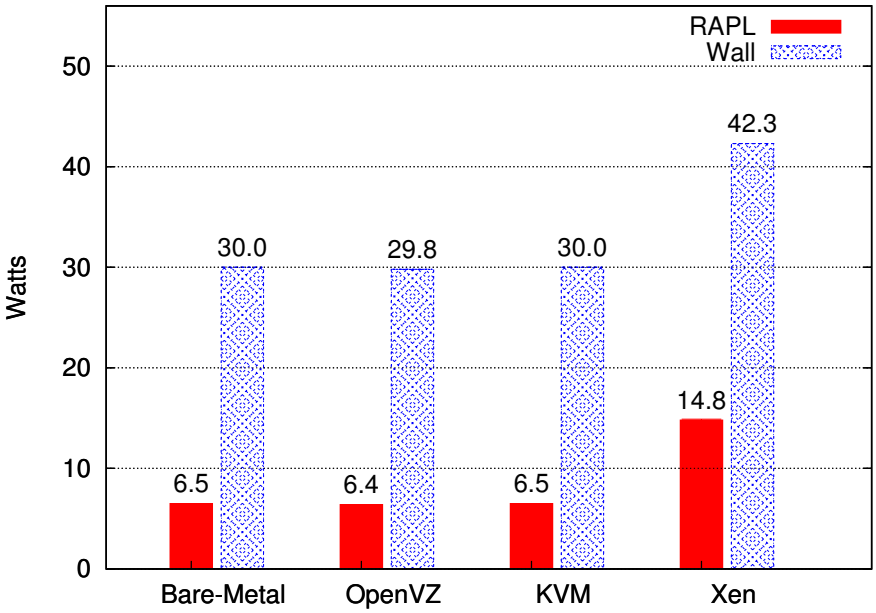


Figure 3.1: Power Consumption - Idle

watts at the wall, and about 6.5 watts at the CPU. The Xen system has a greater external and internal power consumption, with an average of 42.3 watts for the whole system and 14.8 for the CPU. For all of these systems, the error bars, which represent the standard deviation between tests, are very small. It is not surprising that KVM, OpenVZ, and the bare metal system have similar power consumption while idle, since they all take advantage of the standard Linux power saving system. However, in Xen, even the driver domain Domain-0 is in fact a virtual machine, which does not allow the Linux kernel to properly manage the power of the system. Using the Xen power management command "xenpm", we have determined our idle Xen system never enters a sleep state deeper than c1. Our investigation leads us to conjecture that the Xen 4.1 hypervisor does not properly utilize the Core i5 processor's advanced *c-states* (sleep states). In the standard Linux kernel, when a processing core is idle, the system puts the core to sleep. We have forced Xen to use the c-states, yet it does not seem to enter the deep sleep states such as the c7 state, which leads to a much higher idle energy consumption. This was observed despite the fact that we had passed the required boot time parameters to the Xen kernel and enabled the deeper sleep states in Xen power management module.

3.2.2 Power Consumption with Virtualization

Figure 3.2 and Figure 3.3 show the power consumption of our systems while running the Iperf TCP sending and receiving tests. In this experiment, all systems maintained an average TCP transfer rate of approximately 940 Mb/s while both sending and receiving. For

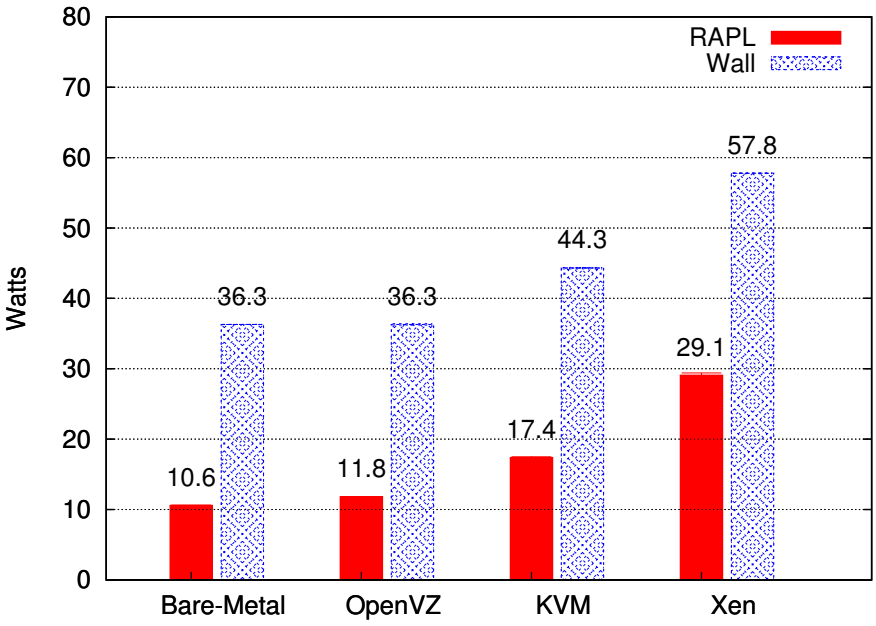


Figure 3.2: Iperf - Sending 1000 Mb/s

the Iperf sending test, the Bare metal system has the lowest power consumption with an average of 36.3 watts consumed for the total system and an average of 10.6 watts for the CPU. OpenVZ has a similar power consumption as the Bare metal system for the total consumption, but a slightly higher internal power consumption of 11.8 watts. KVM has a significantly higher consumption with an average of 44.3 watts external and 17.4 watts internal. Xen has the worst performance with the highest power consumption: 57.7 watts for the whole system and 29.1 watts for the CPU. For the Iperf receiving test, with both Bare metal and OpenVZ, the results are similar but slightly better than for sending. Bare metal has slightly less power consumption with 35.6 watts external and 10.0 watts for internal consumption, as does OpenVZ with 35.5 watts external and 11.2 watts internal. KVM and Xen virtualization each performed slightly worse than on the sending test, with KVM consuming 46.7 watts external and 19.2 watts internal, and Xen consuming 58.1 watts external and 28.1 watts internal. Again, the standard deviations between the tests on the same system are very low. It is not surprising that both KVM and Xen consume considerably more power in these tests since network interface virtualization is a computationally expensive operation [57].

The results of Apache2 Many Client HTTP Download are shown in Figure 3.4 for each system. Our Bare metal control consumed an average of 37.9 watts for the whole system on this test, and an average of 12.1 watts for the CPU. Of the virtualized systems, OpenVZ again performed the best, with an average of 39.1 watts total consumption and 14.3 watts CPU consumption. In this test, KVM and Xen performed similarly, with KVM consuming an average of 52.4 watts external and 24.2 watts internal, and Xen consuming an average

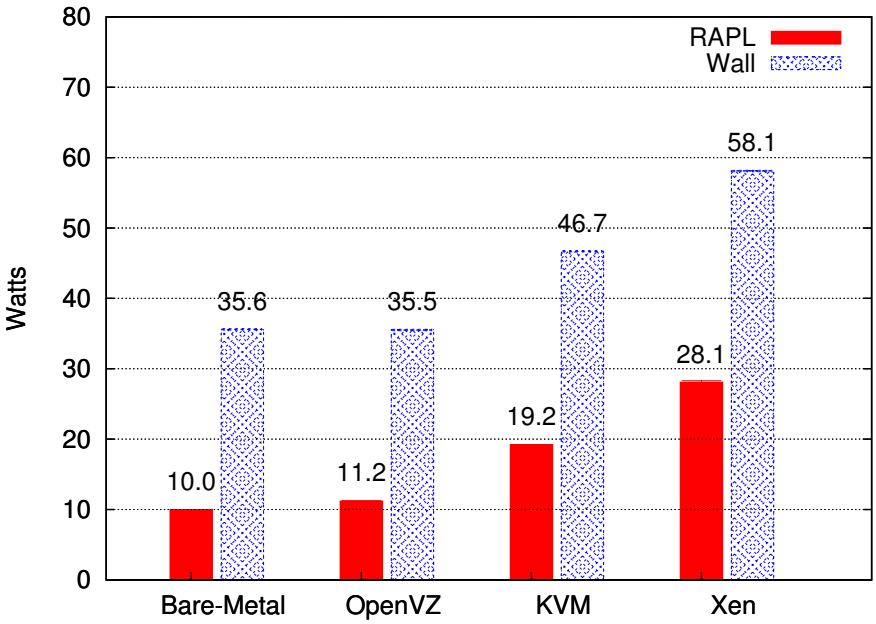


Figure 3.3: Iperf - Receiving 1000 Mb/s

of 54.2 watts external and 26.6 watts internal. The reason Xen and KVM have much closer energy consumption in this test is because both Xen and KVM appear to be load-balancing the apache HTTP threads over all available CPUs. Since all 4 cores now contain active threads the benefits of KVM's ability to reach deeper sleep states is much less pronounced. The standard deviations in this test are slightly higher, though remain small. These results are not unexpected, since transmitting data over a TCP based HTTP connection contains many small control messages and packets. Processing this large number of packets causes higher overhead in virtual machines since virtual packet processing is a much more expensive operation.

3.2.3 Understanding Where Power is Consumed

To understand why there is increased energy consumption and overhead in virtual machines while processing the network tasks described above, we now take a close look at a state of the art virtualized network interface architecture. We use KVM as a representative because of its wide deployment, and because the VirtIO driver it employs is designed to work with a variety of virtualization systems, making the observations widely applicable.

We start our discussion with an overview of KVM's virtual packet processing. Figure 3.5, gives the typical path of a packet entering a virtualized system. The packet is first handled by the physical NIC, which copies the packet in the memory space of the host physical machine and alerts the physical machine of the incoming data through the use of a hardware interrupt. The kernel on the physical machine is then scheduled to run and inspects the

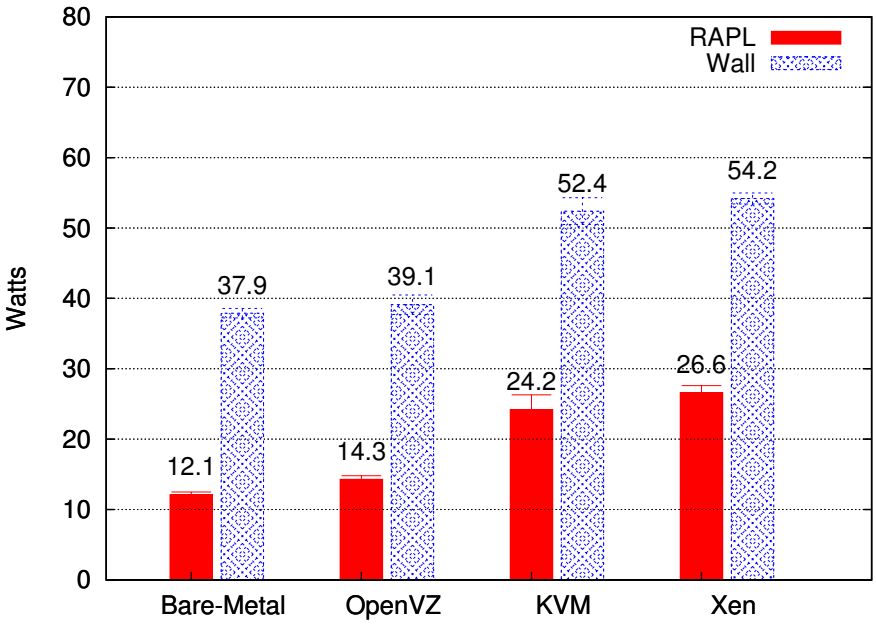


Figure 3.4: Apache2 Download - 100 Clients Downloading

	Bare-Metal	KVM VirtIO
Cycles	11.5M/Sec	51.8M/Sec
LLC References	0.48M/Sec	2.3M/Sec
IRQs	600/Sec	2600/Sec

Table 3.1: Iperf receiving 10 Mb/s TCP traffic

packet. The packet is then pushed through some form of software switch; in our experiments we used a Linux bridge. The switch then sorts the packet and sends it to the virtual NICs back-end, which in the case of KVM is a network tap device. The kernel then notifies the Virtual Machine's hypervisor process of the incoming packet(s) through a software interrupt, and the KVM hypervisor is scheduled to run. The KVM hypervisor process then copies the packet from the host's memory space into the virtual machine's memory space, and sends an interrupt to the virtual machine. Finally, the virtual machine's kernel collects the packet from the virtual NIC and passes the packet to the networking layer. When sending, the virtual machine simply pushes the packet along in the reverse direction through these steps. All of these additional steps cause a virtual machine to consume much more resources than a bare-metal machine when processing network traffic, thus potentially consuming much more energy.

The best way to show the increase in resource consumption created by virtualization is a small experiment. To this end, we designed an experiment to measure the virtualization overhead, and therefore the excessive power consumption, of sending and receiving on both virtual and hardware network interfaces. We once again used the `Iperf` network benchmark

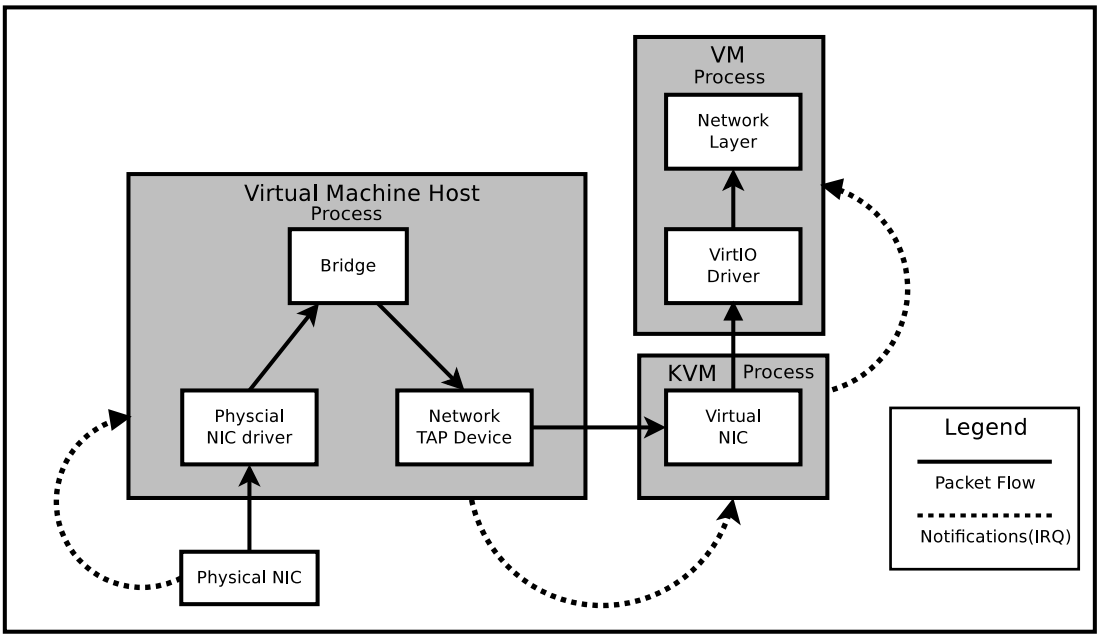


Figure 3.5: KVM Network Architecture

	Bare-Metal	KVM VirtIO
Cycles	4.1M/Sec	33.4M/Sec
LLC References	0.13M/Sec	1.3M/Sec
IRQs	180/Sec	1700/Sec

Table 3.2: Iperf sending 10 Mb/s TCP traffic

to create the TCP traffic to and from a remote host in the same subnet. We used the Linux hardware performance analysis tool `Perf` to collect system level statistics such as processor cycles consumed, last level cache references (LLC), and interrupt requests. We collected statistics for all cores in our physical system, thus our data shows resource consumption both inside and outside the VM. For each experiment, we instructed `Perf` to collect five samples each with a duration of 10 seconds and then averaged them. We tested two systems: our bare metal Linux host, and KVM with VirtIO drivers. The results for the receiving experiment are given in Table 3.1 and the results for the sending experiment are given in Table 3.2. Both CPU cycles and LLC references are given in millions per second. The interrupt requests are given in number per second. We can see that, in the receiving experiment, KVM with VirtIO takes nearly 5 times more cycles to deliver the packets to the VM as the bare-metal host. KVM with VirtIO is also much less efficient on cache than the bare-metal system. This is due to the fact that the VM's hypervisor must copy each packet from the memory space of the host to the VMs space. These copies use up valuable processor cycles and can also evict data from the processor cache. Next we look at IRQs that are used by the physical device to notify the kernel of an incoming packet, and by the

hypervisor to indicate to a running VM that it has received packets. Since KVM also uses interrupts to indicate to a running VM that it has received packets from the network, it comes as no surprise that they would be considerably higher than the bare-metal baseline. As such, if the number of interrupts can be reduced, we can amortize the cost of network processing by batch processing packets, thus reducing the amount of memory access and CPU cycles consumed by the VMs, which would in turn improve the power consumption for many network applications.

3.3 Reducing Power Consumption in KVM

The results of our profiling experiment show that virtualization systems introduce considerable overhead while processing packets. From our previous description of how packets are processed in virtual systems, we propose that buffering packets will help reduce the overhead caused by the virtualization system, and therefore reduce the energy consumption.

It has been established in other contexts, such as mobile devices, that buffering sending and receiving tasks has an energy conservation benefit [22]. Much of this energy saving is due to the efficiency of batch processing of network packets and powering down the radio between network bursts. However, our novel approach in applying packet buffering to virtual packet processing achieves its energy savings by greatly increasing the efficiency with which the hypervisor handles network traffic. Further, in [57] through hardware profiling techniques, we found that virtualized systems use more memory, cache and CPU cycles to process packets than non virtualized systems do. We implement our modifications in the VirtIO drivers of the KVM system. The choice of KVM and the VirtIO drivers is motivated by the fact that VirtIO is designed to be a driver system that is compatible with many virtualization systems. Thus, our modifications to the VirtIO drivers are likely applicable to many other virtualization platforms.

3.3.1 Buffering Timer for VirtIO

We first modified the VirtIO driver to buffer incoming packets, delivering them in bulk to the virtual machine. To accomplish this, we carefully inspected the VirtIO's virtual NIC device code and changed how the VM is notified that it has incoming packets. Instead of immediately sending an interrupt on packet arrival, the KVM hypervisor can now set a timer. When the timer expires the VM is notified of the original packets as well as all other that have arrived since the timer was started.

Although we can configure the VirtIO virtual NIC device to ignore the interrupts created by the VM's transmitted packets, it is not enough to reduce the sending overhead of the VM. This is because the VirtIO drivers inside the VM perform a queue "kick" operation for every buffer of sent packets. The kick operation not only raises an interrupt to the hypervisor, but also performs an expensive locking operation to copy over the packets into

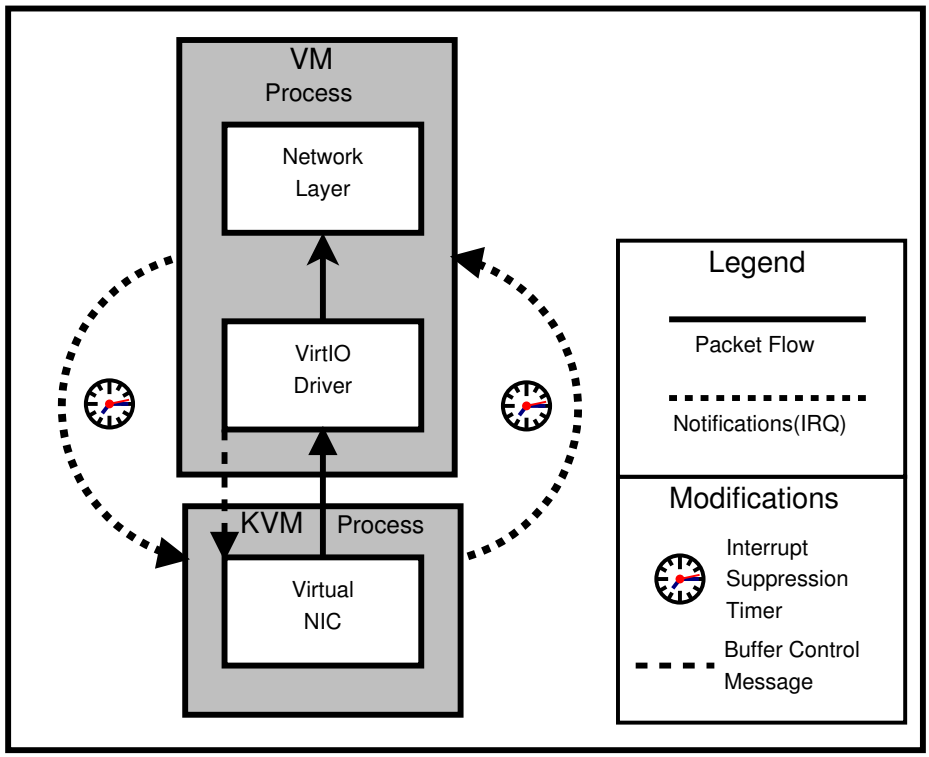


Figure 3.6: Modified VirtIO Drivers

the hypervisor's sending buffer. Thus, modifying the VirtIO's virtual NIC device is not enough to buffer the transmitted packets from the VM, and we must also optimize the driver residing inside the VM.

With this observation, we modified the VirtIO drivers to set a timer instead of immediately performing the kick operation. When the timer expires, a single efficient copy of all the packets from the VM's sending queue is made to the hypervisor. Also, a single interrupt is sent to the hypervisor instructing it to process these packets. Figure 3.6 shows the location of our modifications in the VirtIO NIC structure.

3.3.2 Synchronizing Receiving and Transmitting Delay

After modifying the VirtIO virtual NIC subsystem to allow buffering of transmitted and received packets, we next need to develop a system to synchronize the Receiving (RX) and Transmitting (TX) delays. Synchronization of the delays is needed for a number of reasons. First, we must be able to advertise to the KVM hypervisor the appropriate time to buffer the received packets. Second, the hypervisor must be instructed to only buffer the packets for systems running applications that can tolerate the buffering delay. Finally, to ensure a stable round trip time (RTT), the hypervisor and VM must balance their buffer times. To this end, we developed and implemented a new virtual "hardware" control message, which can be sent from the VM to the virtual NIC device. The control message specifies if the VM

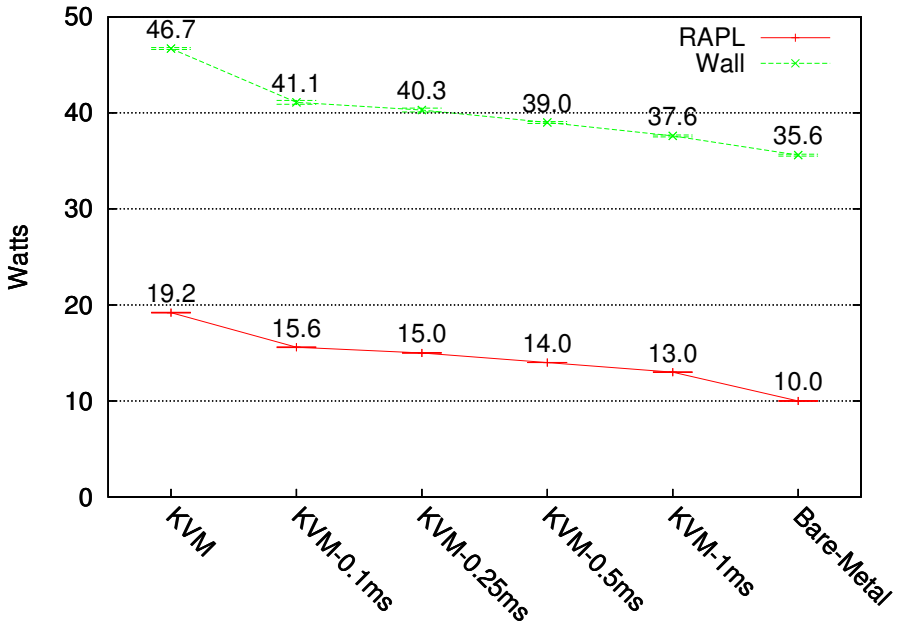


Figure 3.7: Iperf Receiving 1000 Mb/s Improvement

expects the hypervisor to buffer incoming packets as well as how long to buffer the packets for. Our modification allows for each VM on a physical host to enable/disable and control its own buffering independently. Figure 3.6 shows the path of hardware control message as well as a high level overview of our modified driver system.

3.4 Power Savings with Driver Modification

Using the VirtIO modifications and packet rate calculation formula described previously, we once again measure the energy consumption of network tasks on our test platform. We installed our modified drivers in our VM. For all experiments, we once again run them three times and give the average as well as express the standard deviation as error bars on our graphs. We test different buffer times ranging from 0.1 ms to 1 ms, which, as compared to the typical wide area network RTT, is generally negligible.

Figure 3.7 shows the results for receiving of running the Iperf TCP benchmark using our modified KVM driver. In all tests the modified drivers achieve a maximum throughput of approximately 940 Mb/s. Even with a 0.1ms buffer there is an approximately 12% drop in energy consumption from the external Wall measurement, from 46.7 watts to 41.1 watts. With steadily bigger buffer sizes we continue to see steady improvement, up to an improvement of 19.5% over the unmodified KVM. For the internal RAPL measurement, we also see a steady improvement with increased buffer size, from an 18.8% improvement in energy consumption with a 0.1ms buffer to a 32.3% improvement over the unmodified driver with a 1ms buffer. Figure 3.8 shows the Iperf TCP benchmark results for sending. In the

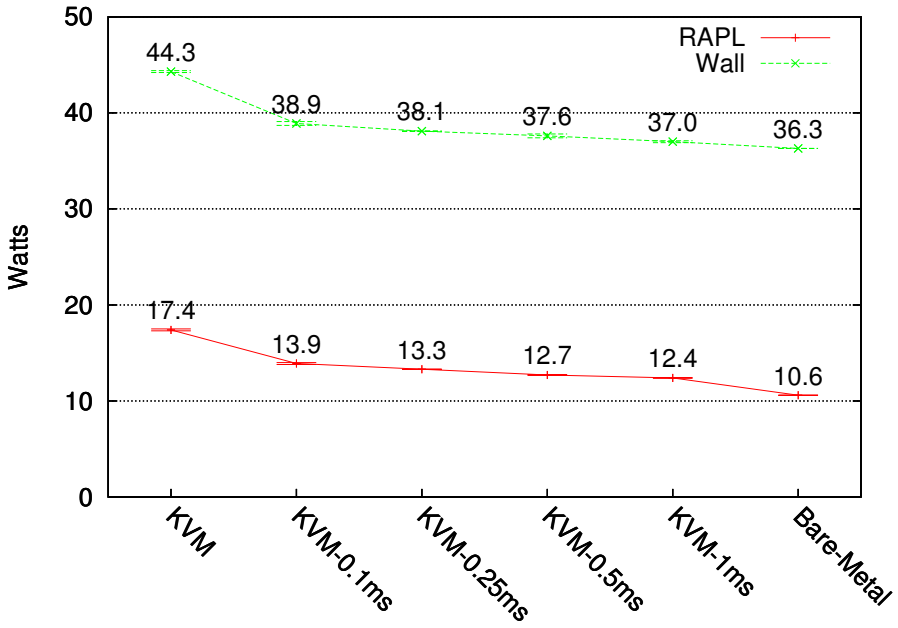


Figure 3.8: Iperf Send 1000 Mb/s Improvement

wall measurements, with the smallest buffer size, there is already a major improvement of 12.2%, from 44.3 watts with the basic KVM driver to 38.9 watts with our modified driver. There is a gradual but steady improvement with increased buffer size, up to 16.5% change in energy consumption with a 1ms buffer. The RAPL measurements show a very similar trend. With a 0.1ms buffer there is a 20.1% change in energy consumption, and with a 1ms buffer we see a 28.7% improvement with our modified KVM driver.

In Figure 3.9, we see the energy consumption of running the Apache benchmark with the basic KVM driver compared to with our modified driver using different buffer sizes. For the wall measurements, there was a fairly steady increase with increasing buffer size. With the smallest buffer size we only saw a 5.7% improvement over the basic KVM driver, however this increased to a 12.4% change in energy consumption with a 1ms buffer. The results for the internal measurements are similar. With a 0.1ms buffer we see a 6.2% change, from 24.2 watts with the basic driver to 22.7 watts with the modified driver. With a 1ms buffer we see a 17.0% improvement over the unmodified driver. It is important to note that our modifications did not reduce the data transfer rate of the HTTP downloads, thus the modified KVM system still served the files in the same amount of time as the baseline systems.

3.5 Impact to Complex Network Applications

So far we have focused on the basic network transactions. We now establish the power consumption profiles of advanced and more realistic network applications, as well as the

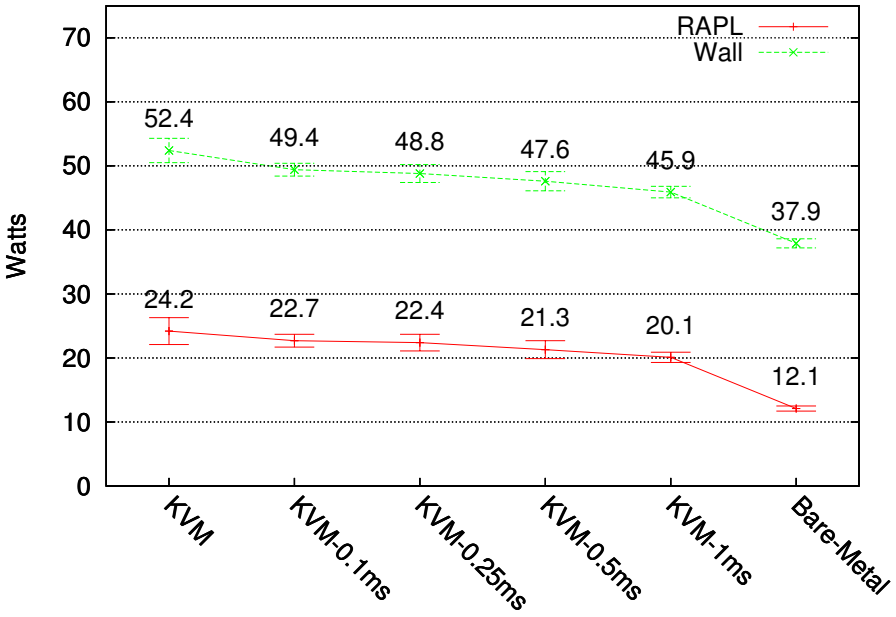


Figure 3.9: Apache2 Download Improvement

impact of buffering. Although the simple benchmarks presented previously illuminate some important characteristics of virtualized systems in terms of energy consumption, it is critical to examine more complex applications which are more commonly found in real world datacenters.

3.5.1 Benchmark Setup

RUBBoS Bulletin Board Benchmark: To further understand the overall system power consumption, we have devised a comprehensive benchmark based on a simple 2-tier web server and database. We used the Debian repositories to install the `Apache` 2.2 Web Server and the `MySQL` Server 5.5. To create a web application representative of a real-world service, we installed the `RuBBoS` bulletin board benchmark. We chose the PHP version of the RuBBoS and installed the necessary Apache extensions for PHP. We then installed the RuBBoS data into our MySQL database.

Although RuBBoS comes with its own client simulator, we used the one from the Apache benchmark instead, which has been more commonly used for web server stress testing. Also, we only require the maximum request rate, which is more straightforward to extract with the Apache Benchmark. We ran the Apache Benchmark against the RuBBoS website in each of the test setups. We emulated 100 clients requesting the latest forum topics page. By using this page, the web server must perform a single SQL query and render the PHP page for each user request. We then used the Apache benchmark to calculate how long it takes to service the 500,000 requests.

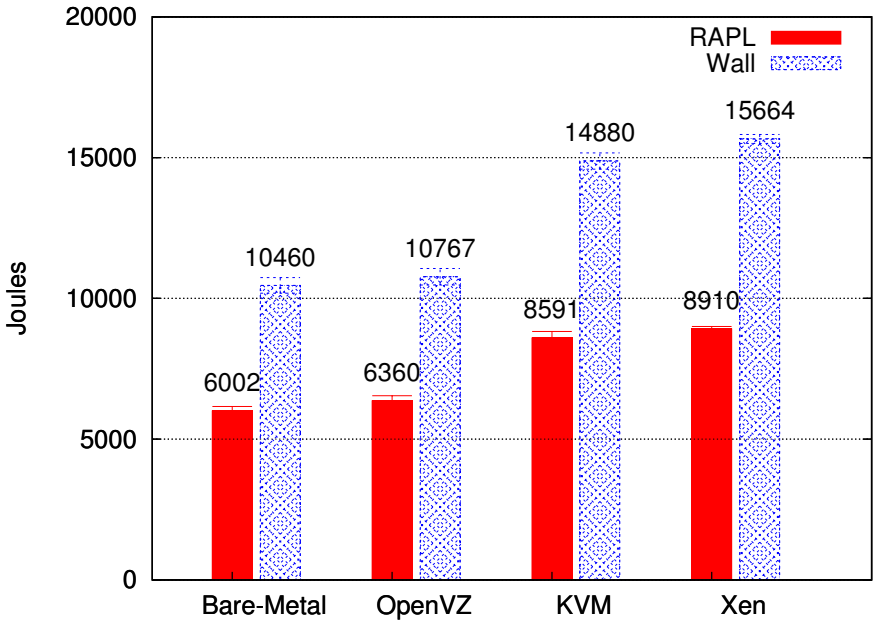


Figure 3.10: RUBBoS - 100 Clients, 500,000 requests

Tbench Network File System Benchmark: Our final experiment employs Tbench, which emulates the network portion of the standard Netbench performance test. Tbench sends TCP data based on a workload profile, which simulates a network file system. We specified our test system as the Tbench server and then configured our client emulator to simulate 100 processes reading and writing the remote file system.

3.5.2 Power Consumption

Our results for the RUBBoS benchmark, the comprehensive web server test, are shown in Figure 3.10. The Bare metal system consumed an average of 10460 joules total and 6002 joules for the CPU. OpenVZ was comparable, with an average consumption of 10767 joules for the whole system and 6360 joules for the CPU. KVM and Xen performed worse, with KVM consuming an average of 14880 joules total and an average of 8591 joules for the CPU, and Xen consuming an average of 15664 joules total and an average of 8910 joules for the CPU. Unlike the previous benchmarks we have performed, the RUBBoS benchmark is not strictly a network-bound application. This multi-tier benchmark has to render a PHP page as well as perform database look ups to service each incoming request, both of which are hard on memory and CPU resources. This leads to even higher energy consumption because not only are the network accesses causing virtualization overhead, but the access to memory by the database is also contributing to the virtualization overhead.

Figure 3.11 shows the power used by each of our systems running the Tbench benchmark test. The Bare metal system uses 43.5 watts for the total system measured at the wall, and

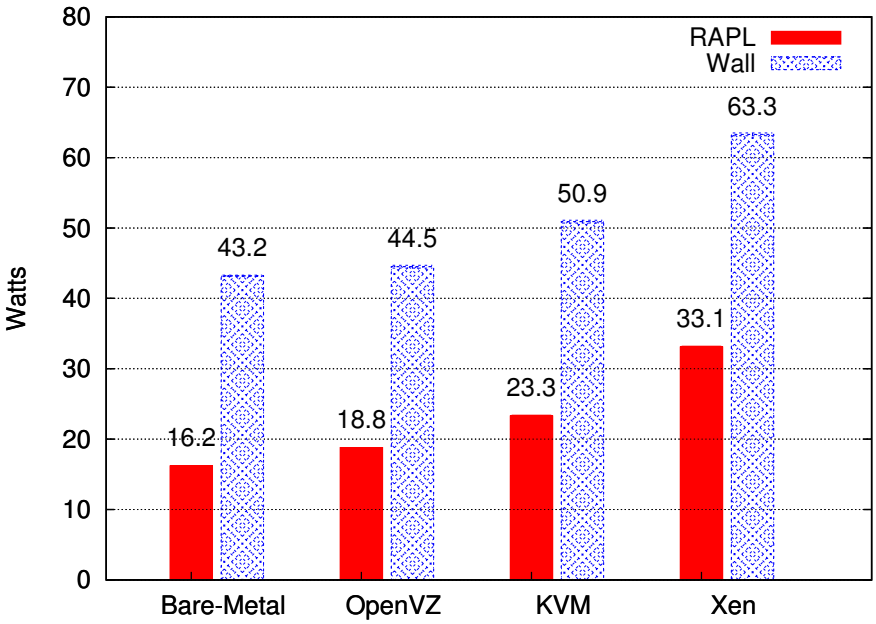


Figure 3.11: Tbench, 100 network processes, read/writing remote file-system

16.2 watts for the internal CPU. OpenVZ performs nearly as well, consuming 44.5 watts at the wall and 18.8 for the RAPL counter. KVM also performs fairly well on this test, using 50.9 watts externally and 23.3 watts for the CPU. Finally, Xen uses 63.3 watts for the total system and 33.1 watts for the CPU. In this benchmark the Tbench server must emulate the responses to the read and write file system requests generated by the remote clients, while at the same maintaining high network throughput. This is a more complex use of the network communication infrastructure as performance is dependent on the Tbench server processing the clients' requests quickly.

3.5.3 Impact of Buffering

Using our modified drivers VirtIO drivers described previously, we test the candidate buffer time of 0.1 ms, 0.25 ms, 0.5 ms and 1 ms. For all experiments we once again run them three times and give the average as well as express the standard deviation.

Figure 3.12 shows the improved energy consumption while running the rubbos benchmark, comparing the unmodified KVM driver to the modified driver at our specified buffer sizes. The external measurements taken from the wall show that even with the smallest buffer size, there is a significant improvement in energy consumption. With a 0.1ms buffer, there is a 11.4% drop in energy consumption, from 14880 joules with the unmodified driver to 13181 joules with the introduction of the buffer. There is then a steady improvement with increasing buffer size, up to a 16.5% change from the unmodified KVM with a 1ms buffer. The internal RAPL measurements are similar, with the major improvement occurring with

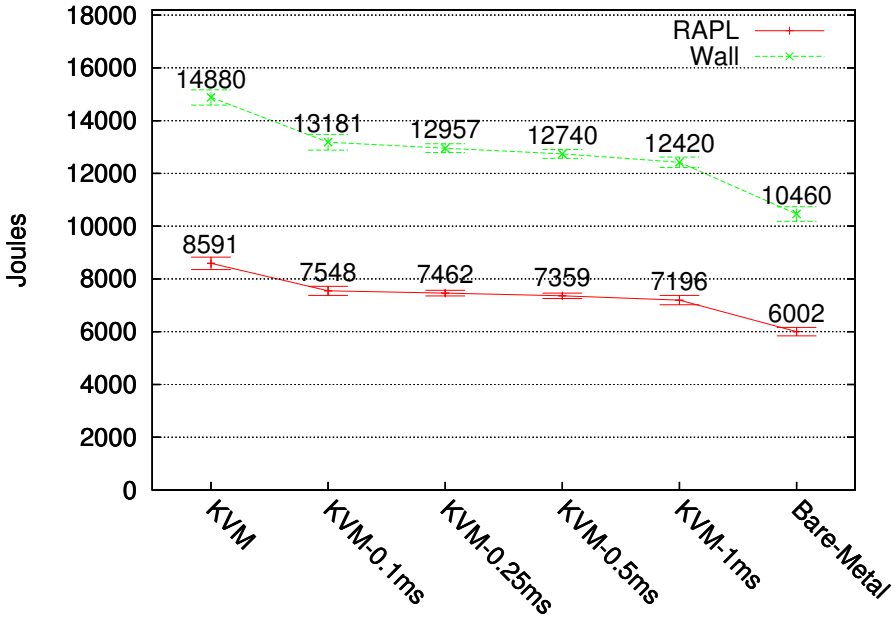


Figure 3.12: RUBBoS Improvement

the introduction of even the 0.1ms buffer size, with a 12.1% drop in joules consumed. There continues to be an improvement in energy consumption with increasing buffer size, with our largest buffer size, 1ms, resulting in a 16.2% improvement, from 8591 joules using the standard KVM driver to 7196 joules with our modified driver. The modified drivers were not only able to reduce the energy consumption of our virtualized system but actually increased the performance as well. With a buffer of 1.0 ms our modified KVM virtual machine was able to service nearly 10% more request per second than the unmodified KVM system. As stated previously, this benchmark requires many subsystems in addition to the network, leading to even higher overhead due to non-network causes. However, we show that in this complex case, we still manage the impressive energy reduction of up to 16.2%, while improving performance. This improvement in performance is because the system resources are now freed to do useful work inside the VM such as answering queries or rendering PHP pages.

In Figure 3.13, we present the results of the Tbench test, comparing the power consumption of the Bare metal system to the unmodified KVM system and to a KVM system running the modified driver with varying buffer sizes. The unmodified KVM system consumes 50.9 watts at the wall, and 23.3 watts at the CPU. Running the modified KVM driver with a buffer of 0.1ms, the power consumption is 49.5 watts at the wall, and 23.1 at the RAPL counter. With a larger buffer size, the power consumption improves, down to 49.0 watts at the wall and 22.8 watts at the CPU with a buffer size of 0.25ms. With a buffer size of 0.5ms, the system consumes 47.6 watts at the wall and 22.0 watts at the CPU, an improvement of 6.5% and 5.6% respectively over the unmodified KVM. The largest buffer size we test, 1ms,

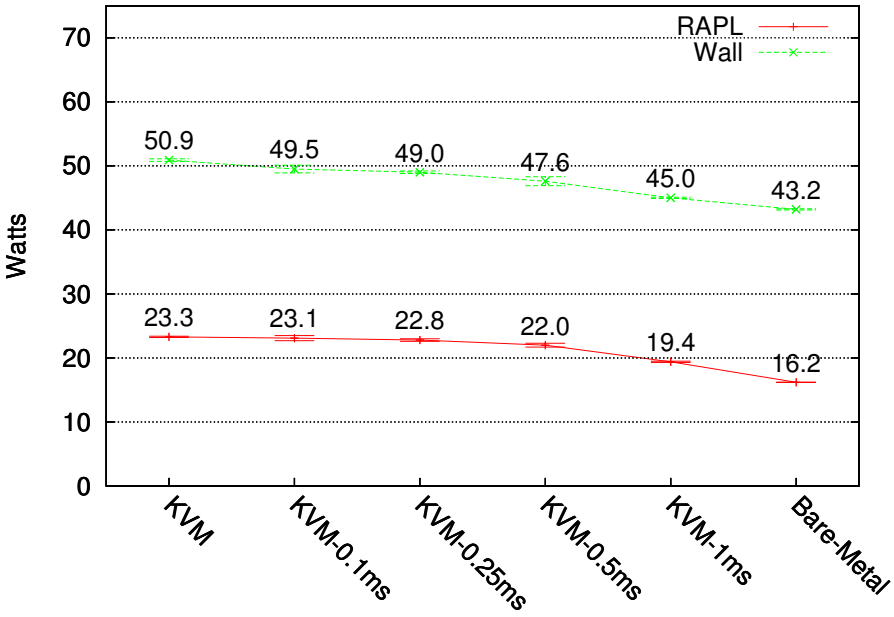


Figure 3.13: Tbench, 100 network processes Improvement

does have lower power consumption but unfortunately it suffers from an approximately 6% decrease in throughput. However, the buffer sizes 0.1ms 0.25 ms, and 0.5 ms have identical performance to their KVM base-line. The loss of performance with Tbench as our buffer sizes approaches 1ms is likely due to the fact that network file systems are more susceptible to increases in latency than our web server benchmark RUBBoS, for example. Tbench serves to illustrate that many applications benefit from this buffering technique, however the amount of buffering to ensure optimal performance of the application could vary.

3.6 Discussion

In this chapter we showed, through both external power meter measurements and internal RAPL hardware profiling counters, the energy overhead created by using virtualized systems. We found that due to virtualization overhead, hypervisor based virtualization systems such as KVM and Xen can consume considerably more energy when running typical network tasks. On the other hand, we find that the container virtualization system OpenVZ consumes near identical amount of power as our non-virtualized baseline system.

Our experiments with other complex network applications, such as the distributed memory object caching system `Memcached` and Online Transaction Processing (OLTP) benchmarked by `sysbench`, their energy consumption does decrease but at the cost of a slight loss of performance. After inspecting these two systems, we conjecture the following reasons. First, both applications' benchmarks appear to employ a stage that incorporates blocking I/O, meaning that any delay in response will slow down the process of the entire bench-

mark. Second, both applications are sensitive to increases in latency, especially memcached, which is designed to be a low latency memory object caching system. However, it is likely that non-blocking I/O implementations of these applications would also see improvement in energy consumption.

Further, our modifications to KVM's VirtIO driver are likely compatible with recent advances in VM packet switching such as the VALE [55] software switch. For a future work we plan to analyze the energy consumption of these advanced packet switching techniques as well as test their performance with our VM packet buffering techniques.

We show that it is possible to conserve energy without loss of performance for many typical networked applications through the use of packet buffering. Our real world practical modifications show that a busy virtualized web server can reduce its energy consumption by over 16% by buffering its incoming and outgoing packets. However, it remains to be discovered what other networked applications can benefit from our optimizations.

Chapter 4

Network Performance of Virtual Machine Based Cloud Environments

In most cloud computing platforms, virtual machine (VM) technology is the core technology enabling a single physical machine to be partitioned into multiple logical systems. By this means, cloud providers can decrease operational costs through server consolidation and better utilization of computational resources. Cloud users can also easily outsource their computational requirements and purchase suitable cloud instances accordingly. The benefit of VM technology, however, comes together with overhead, affecting virtually every aspect of the computer system, especially the network subsystem. For example, it is reported that current network communications on VMs (e.g., Amazon EC2 instances) may experience significant throughput instability and abnormal delay variations [68] . Despite this pioneering research many issues remain unexplored. For example, existing research has mainly focused on performance issues due to network instability in low latency environments inside the cloud. The effect of this instability on the performance of network protocols running over higher latency, wide area networks has seldom been explored. Also, the effect of computational load inside a cloud virtual machine and its effect on network performance and stability has yet to quantified. It has been shown, in the vSnoop developed in Purdue University, that the TCP performance with Xen virtulazation can be improved by offloading TCP acknowledgements to the driver domain [36]. We however show that not only does TCP performance degrade when the physical system is under heavy use but the packet drop rate can increase by up to 20%, indicating TCP acknowledgement offloading alone is not enough to fix the underlying performance issues. More recent research has been conducted into packet delay in virtualized system hosted on single core physical machines [70]. It remains to be determined what other factors contribute to network delay in cloud virtualized systems when they are hosted in a more complex multi-core environment.

In this chapter, we systematically investigate the virtualization overhead on network performance in VM based cloud platforms, striving to not only understand the performance degradation and variation phenomena clearly but also dive deeply to uncover the root causes. To this end, we take the Xen hypervisor based cloud Amazon EC2 as a case study, conducting extensive measurements on the network performance of its large instances (which are advertised as being provisioned with enhanced I/O performance). Our measurement results show that the network performance degradation and variation phenomena can be prevalent and significant even within the same data center. Taking TCP traffic as an example, among 10 instances we randomly pick from Zone A in Amazon's Oregon data center, up to 8 of them have experienced certain levels of throughput degradation (with the maximum up to 50%) even when the VMs' virtual CPUs are nearly idle. When the VMs are fully utilized with computation tasks, the throughput degradation can be as much as 87%. For UDP traffic, the throughput can also fall about 19% when a VM is heavily utilized and the packet drop rate can increase to nearly 20%. In addition, we find that the round trip time on a VM fully utilized with computation tasks can rise from around $10ms$ to as high as $67ms$.

To fully understand the root cause of these phenomena, we conduct detailed system analysis on Xen's CPU scheduler and network architecture. Our analysis reveals that the network performance variation and degradation are mainly due to the "interference" between the CPU utilization for computation and network communication, causing the network communication to be starved for CPU. In particular, the "netback" process of a VM in the driver domain cannot be scheduled on a CPU in time, which causes the arrived network packets to stay in the receiving buffer of the driver domain for an extra period that does not exist in a normal non-virtualized machine, quickly overflowing the buffer and causing later arriving packets to be dropped. Moreover, we identify the two typical scenarios that lead to such a situation, namely, "co-runner interference" and "self interference". The former scenario mainly happens among different VMs, where a VM or the netback process of a VM is starved due to waiting for the scheduling of other VMs. And the latter scenario happens within a VM, where the netfront process of the VM is starved because other processes (mainly computation processes) have used up the CPU resources allocated to this VM.

To verify our analysis, we further set up a Xen based cloud environment similar to Amazon EC2 on our local cloud testbed and conduct extensive experiments to carefully examine our findings. The results match well with what we have observed directly from the Amazon EC2 platform and confirm the validity of our analysis.

Based on our findings, we propose a solution to improve and stabilize the network performance. We develop a scheme to schedule the VMs mainly with computation tasks and the VMs mainly with network communication tasks to separate CPU pools. This way VMs with mainly network communication tasks can achieve excellent network performance while still keeping reasonably good computation performance, without disturbing and degrading

the performance of the VMs with mainly computation tasks. We conduct an evaluation on our cloud testbed. The results further demonstrate the effectiveness of our solution.

In addition to the works already cited the following research has been conducted on cloud network performance. Researchers at Duke University and Microsoft Research have recently proposed a frame work for comparing public cloud providers titled *CloudCMP* [42]. CloudCMP allows organizations to compare public clouds on metrics such as computational performance, price, and storage offering. Ostermann *et al.* [48] evaluated Amazon EC2's suitability for high performance scientific computing. The researchers found that the use of virtualization can impose significant performance penalties on many scientific computing applications. In 2011, Shieh *et al.* [64] discussed the network performance issues specific to multi-tenant data centers. The authors introduced *Seawall*, a network bandwidth allocation scheme, which can fairly share the data-center's resources according to administrator policies. Further research related to network sharing in cloud based multi-tenant data-centers was preformed by Popa *et al.* [50]. The authors investigated issues related to minimum guaranteed and proportional bandwidth sharing in a cloud computing context. A set of bandwidth allocation strategies were proposed to both improve performance and fairness of these systems. There have also been many performance analysis testing the suitability and performance of virtualized system for specific applications and scenarios. [30][18][31][33].

4.1 Network Performance in VM-based Clouds

Our measurement goals are to find out how network bandwidth, delay, and stability are affected by the variations of CPU loads. Further, we want to discover what differences in network performance exist between instances of the same class, performing the same task, in the same data center. To ensure an accurate and representative analysis we collect a wide arrangement of samples from cloud instances of the same class. Also, to ensure accuracy our experiments must not rely on a single protocol. To this end, we devise the following cloud *instance selection methods* and *measurement settings*.

For our cloud instance selection we choose the current industry leader in cloud computing Amazon EC2 as our cloud provider. We choose Amazon EC2 not only because it is the current market leader but also because of its heavy use of the open source Xen hypervisor to provide virtual machines. Amazon's use of Xen allows us to closely investigate performance issues by looking into the architecture and source code. Also, Xen is used by numerous other large cloud providers such as Rackspace [53], making our findings more general and applicable to other clouds. To accurately measure network performance, we provision Amazon EC2 “Large" instances. The choice of Large instances is motivated by two key factors. First, they provide ample computation and memory resources for most web applications including our benchmarks and experiments. Second, they are rated to have “high" I/O capabilities, making them a natural choice for bandwidth intensive applications. The choice

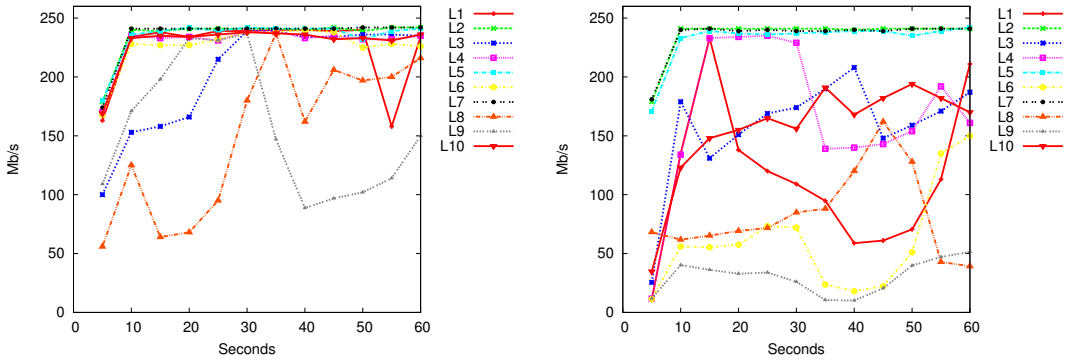
of large instances also allows our results to be more general since smaller instances may simply not be provisioned with enough resources to complete our benchmarks. It is also expected that smaller instances would show the same behaviors or worse since they lack much of the resources granted to their larger counterparts.

For our measurements we require the ability to target loads on the network and CPU subsystems. To this end, we choose iperf and sysbench, respectively. Iperf is a widely used configurable network benchmark, allowing a user to gauge the performance of both TCP and UDP data flows. Sysbench is chosen for CPU benchmarking because it is widely used and can be configured to target the CPU using its prime number benchmark. Using these targeted benchmarks we can easily apply certain amount of computation loads to CPU and measure the network performance accordingly. We can then compare the performance across different scenarios and load patterns. Further, the use of these two targeted applications also allow us to view the direct consequence of CPU load on the network protocols, whereas a more complex application may suffer from application specific problems when being migrated to the cloud environment. For all measurements our CPU benchmarks are set to use the lowest Linux scheduler priority using the "nice" command. The use of the lowest nice priority for the CPU benchmark ensures that the network task Iperf will be given priority to run on the VM's virtual CPU. By doing this we ensure that inside the VM the CPU benchmark does not take CPU cycles needed to process our network traffic. For operating system selection we choose Ubuntu 12-04 64 bit, and the packages are installed from the Ubuntu repository.

To fully understand the network performance degradation and variation phenomena, our measurements investigate a wide range of settings, across both TCP and UDP traffic, and for each type of traffic, examining both sending and receiving performance. We also consider different situations such as whether there are computation loads on VMs' CPUs or not. In addition to throughput and packet drop rate, we also measure round trip time by ICMP, striving to identify potential relations among our findings. In the following subsections, we first discuss our results on TCP traffic. We then proceed with UDP traffic and present our round trip time measurement by ICMP thereafter.

4.1.1 TCP Traffic

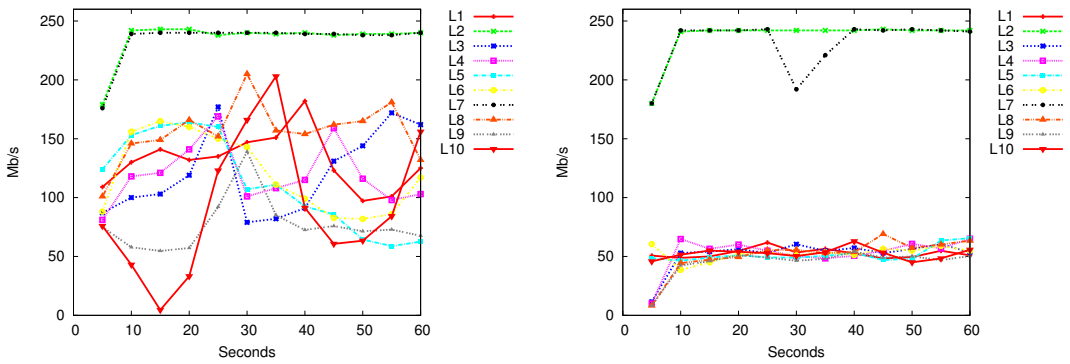
For our TCP measurement, we provision 10 EC2 large instances from availability Zone A in Amazon's Oregon data center. Using 10 instances we will be able to accurately find the bandwidth and delay variation between instances of the same class while the CPU is both *idle* and *fully utilized*. We define idle to be only Iperf and required operating systems services running. Iperf and the operating system consume less than 10% of the system resources while running. Fully utilized is where we also saturate the remaining CPU with our Sysbench prime number benchmark. We also configure a test system outside of the Oregon cloud. The test system has a maximum network throughput of 300 Mb/s and is



(a) TCP Sending Idle (b) TCP Sending Load

Figure 4.1: Ten EC2 Large Instances Sending TCP Traffic

used to send and receive data to and from our cloud instances. We number our EC2 large instances from L1-L10 and test each one sequentially under the following conditions: 1) Maximum achievable sending bandwidth while instances are idle and under computation load, and, 2) Maximum achievable receiving night bandwidth while the instances are idle and under computation load. Iperf is configured to run for 60 seconds and report the bandwidth average every 5 seconds. The tests are run sequentially to minimize the interference from potential changes in network conditions and topology at Amazon's data center.



(a) TCP Receiving Idle (b) TCP Receiving Load

Figure 4.2: Ten EC2 Large Instances Receiving TCP Traffic
bottom

The results from the sending and receiving measurements are given in Figure 4.1 and Figure 4.2, respectively. They show that even with identical specification, large variations in performance can be observed when the systems are under load, and even in many cases, when the systems are idle. We first focus on the sending scenario.

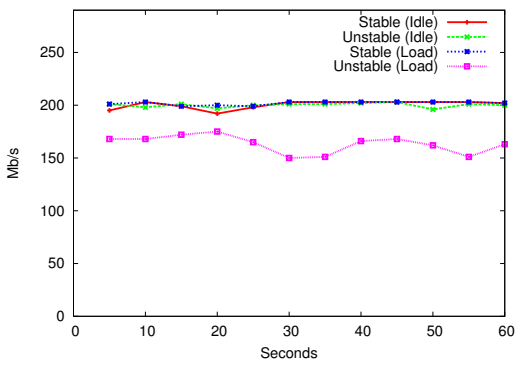
Under the sending measurement, each of our 10 instances sends data using TCP to our remote host. As shown in Figure 4.1 the base line performance of the 10 instances are

relatively similar, with most of the instances reaching a stable sending rate of approximately 240 Mb/s. 3 of the instances however experience significant instability over the 60 seconds of measurement period. One of the unstable instances reaches an average of only 200 Mb/s, a drop of nearly 17% on throughput. The other two unstable instances show even weaker performance at less than 160 Mb/s, a drop of over 30%. Next, we introduce the CPU load using Sysbench and once again send data to our remote host. The results are given in Figure 4.1b. It is easy to see that although 3 of the instances show nearly identical bandwidth to their baseline idle performance, 7 out of 10 instances now show tremendous variation in terms of bandwidth. In particular, 4 of the instances drop over 30% to less than 160 Mb/s. Two instances fall more than 66% to less than 80 Mb/s, and one of the instances falls even lower to 30 Mb/s, an incredible drop of over 87%.

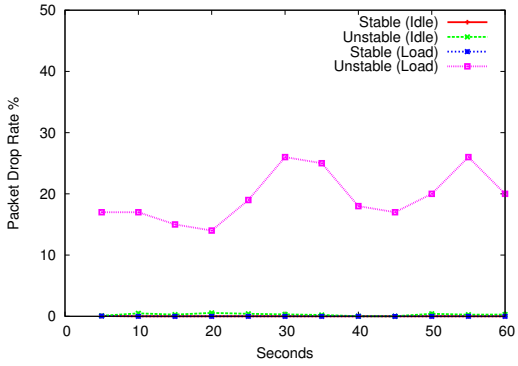
Next we look at the receiving measurements. The baseline performance is given in Figure 4.2a, where 8 out of 10 instances suffer from severe bandwidth instability even when under near idle conditions. The 2 stable instances have a receiving bandwidth of 235 Mb/s. 6 of the unstable instances achieve a performance of less than 150 Mb/s, a considerable loss of over 35% on throughput when compared to the two stable instances. Furthermore, another 2 large instances achieved even lower performance with well under 100 Mb/s, a drop of over 50%. We now focus on the measurements under computation load which can be found in Figure 4.2b. Once again the stable instances from the previous idle experiment remain at nearly the same bandwidth. One instance does temporarily drop its receiving rate but recovers quickly, this is possibly due to a temporary loss of packets. Interestingly, all the unstable instances now stabilize, unfortunately all 8 instances stabilize at only about 50 Mb/s. It is clear that the introduction of computation load on these instances can have a devastating effect on the bandwidth performance of many cloud instances. In addition, receiving traffic can suffer worse than sending traffic in terms of performance loss and variation.

The measurements on TCP traffic already show that even when instances are all the same class, allocated in the same data center, communicating to the same end host, and under the same conditions, there can be considerable differences in terms of network performance. In section 4.2 we will show that overloaded physical hosts, mixed with the high overhead introduced by network interface virtualization, are likely the cause of this issue. Another finding from our measurements is that there are generally two different types of instances. One type is like the 2 instances that are rarely affected by changes in computation load inside the virtual machine. We thus call them “stable" instances. The other type is like the 8 instances that have considerably worse performance when under load. We thus refer to them as “unstable" instances. It is worth noting that preliminary measurements using other EC2 data centers, namely California and Virginia, indicate these issues are not limited to Amazon’ s Oregon data center. For a more detailed investigation on the network conditions such as jitter and drop rate, we now move onto our measurements on UDP traffic.

4.1.2 UDP Traffic

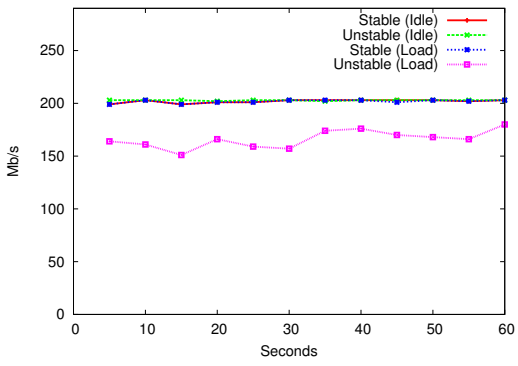


(a) UDP Bandwidth(Receiving)

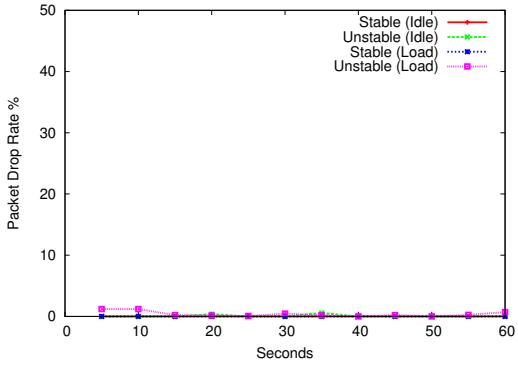


(b) UDP Packet Drop(Receiving)

Figure 4.3: EC2 Large UDP performance Receiving 200 Mb/s
bottom



(a) UDP Bandwidth (Sending)



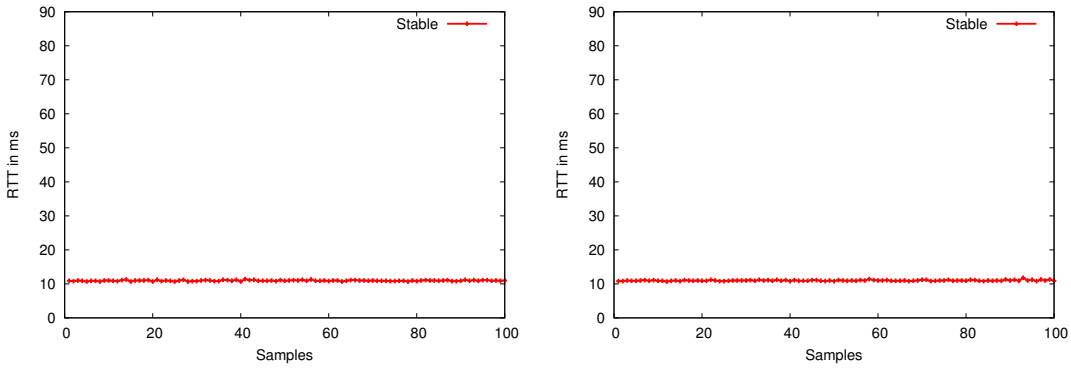
(b) UDP Packet Drop Rate (Sending)

Figure 4.4: EC2 Large UDP performance Sending 200 Mb/s
bottom

Unlike TCP, UDP's stateless best effort packet delivery allows us to directly observe the maximum throughput and data loss rate of our instances. This is because we will not have any issues with TCP flow control and retransmissions.

We select two representative instances from our previous experiment so we have one "stable" instance and one "unstable" instance. We test each instance while sending and receiving at their maximum rate using the same remote system described in the TCP experiment. Since we must select a data rate for UDP we decided on 200 Mb/s, which could easily be sustained by both machines while idle. Once again we use Sysbench running at the lowest Linux nice priority. We show results for data-rate and packet loss rate. Once again each test is run for 60 seconds, and each system is tested while idle and under load.

First we describe the receiving experiments, the total UDP receiving rate is given in Figure 4.3a and the packet drop rate is given in Figure 4.3b. As can be seen while the systems are idle both the unstable and stable instance maintain a receiving rate of 200 Mb/s. When we run our Sysbench CPU prime experiment concurrently with our UDP receiving task the incoming bandwidth of the unstable instance falls nearly 19% to 163 Mb/s. The stable instance on the other hand shows no noticeable drop in bandwidth while under load. We now focus on the drop rates, which can be found in Figure 4.3b. As can be seen our unstable instance increases from a near negligible drop rate when idle to a massive increase of nearly 20% when under load. We will show in section 4.2 that this increase is most likely due to long CPU scheduling delays between a Xen Virtual machine and the VM's "Netback" thread. The Netback thread runs in Dom0 of the Xen system and is responsible for moving packets to and from the Virtual Machine. Since the packets are not delivered to the VM the incoming packet queue becomes full in the physical machine which manifests itself in the large amount of data-loss observed in our unstable instance. This is happening despite the fact that the CPU bound task has been set to the lowest nice priority provided to us by the Linux operating system.

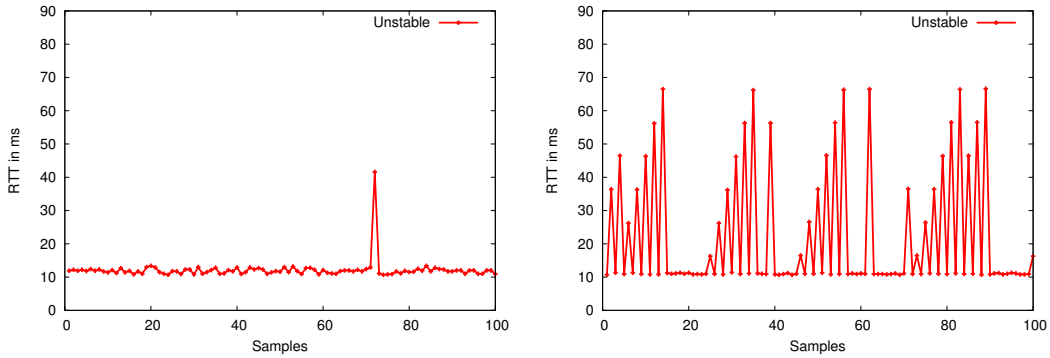


(a) RTT idle (Stable Instance)

(b) RTT load (Stable Instance)

Figure 4.5: ICMP Round Trip Time Stable EC2 Large Instance
bottom

Next we describe the sending experiments; the total UDP sending rate is given in Figure 4.4a and the packet drop rate is given in Figure 4.4b. The throughput remains at the optimal 200 Mb/s for both our instances when no load is introduced inside the VM. However, when we activate our CPU benchmark the performance of the Unstable instances tumbles 37 Mb/s to 166 Mb/s, a drop of over 18%. The stable instance shows no measurable change in outgoing bandwidth. We now turn our attention to the packet drop rate seen in Figure 4.4b. The result shows no increase for the stable instance while under a CPU load and only a minor increase for the unstable large instance. When contrasted with the results of our UDP receiving experiment, we notice that although the data rates for our unstable instance falls by a similar degree, the packet drop rates are very different. In the sending



(a) RTT idle (Unstable Instance)

(b) RTT load (Unstable Instance)

Figure 4.6: ICMP Round Trip Time Unstable EC2 Large Instance bottom

experiment introducing a CPU load on our unstable instance does not noticeably increase the packet drop rate but in our receiving experiment the drop rate jumps to nearly 20%. This observation indicates that while sending, the sending buffer inside the VM becomes full, causing the operating system to reduce the speed of the UDP application. The buffer becomes full because the netback thread is not able to get enough CPU time to forward the packets to the physical network. The reduction in rate means that no measurable increase in drop rates is found. Receiving, on the other hand, overloads a receiving buffer in Dom0 controlled by the netback thread. Since UDP has no native notion of flow-control our remote host continues to send at its full 200 Mb/s and many of these packets are dropped by the physical machine.

4.1.3 Round Trip Time by ICMP

We next devised an experiment to test the effect CPU load had on the round trip time (RTT) packets sent to our cloud instances. Once again we select a representative Stable large instance and a representative Unstable large instance. We use a ICMP echo request packet (ping) with 100 bytes of data and record the amount of time to receive the ICMP echo response from the cloud server. We use a local workstation to perform this test and send an echo request every 100 ms for ten seconds. We use the hping3 tool to create our packets for this experiment. Once again we test the instance while idle and while under a high CPU load generated by the Sysbench CPU prime benchmark. Again we set Sysbench to have the lowest priority. The results for the stable instance are given in Figure 4.5 and the unstable are given in Figure 4.6.

This experiment generated some interesting results. As can be seen in Figure 4.5a and Figure 4.5b the stable instance showed no measurable increase in RTT with the average of both tests being approximately 10ms. As can be seen in Figure 4.6a the unstable instance maintains a RTT nearly identical to the stable instance while idle. However, when the load

is introduced the average RTT doubles to over 20 ms with some packets samples showing an RTT as high as 67 ms.

4.2 System Analysis and Verification

To further examine the root causes of the unstable performance experienced by our EC2 large instances in the previous section, we will now perform a system analysis and confirm our findings on our local cloud. The use of a local cloud is essential to our understanding of the underlining issues, since public cloud computing platforms such as Amazon EC2 do not allow us to monitor the physical machines directly. To this end we devise a set of experiments on our local platform to illuminate what subsystems are causing the massive performance variation and loss experienced in our previous experiments.

4.2.1 System Analysis

To fully understand the root causes of the unstable performance experienced by our EC2 VMs we must look closely at the architecture of Xen. It is widely known that Amazon EC2 uses a version of the Xen server, for which the core source code is available and open-source thus we implement a Xen based cloud system in our local network. Of particular interest is the Xen network subsystem and the Xen CPU scheduler. We have discovered the interactions between these two subsystems leads to the unstable performance found in our previous experiments. We begin our discussion with a analysis of the CPU scheduler subsystem.

4.2.2 Xen CPU Scheduler

The Xen hypervisor supports a number of CPU schedulers such as Borrowed Virtual Time (BVT), Simple Earliest Deadline First (SEDF), and the Credit scheduler. The credit scheduler is the most widely deployed Xen CPU scheduler because it provides load balancing across multiple processors cores and can be configured to be work conserving or not based on the requirements of the system administrator [11]. We conjecture the credit scheduler, or a modified version of it, is likely the scheduler used by EC2. We suspect this for two reasons. First, advanced features such as global load balancing are likely required by EC2. Secondly, the credit scheduler has a fixed 30 ms maximum time slice and as our round trip time (RTT) experiment shows in Figure 4.6b the increases in RTT are approximately multiples of 30 ms.

The Xen credit scheduler provides load-balancing for Xen virtual machines by scheduling Virtual CPU (VCPU) based on a weighted priority and optionally a maximum utilization cap. It should be noted that the use of the maximum utilization cap makes the credit scheduler behave in a non-work conserving fashion, this is because even if there are idle

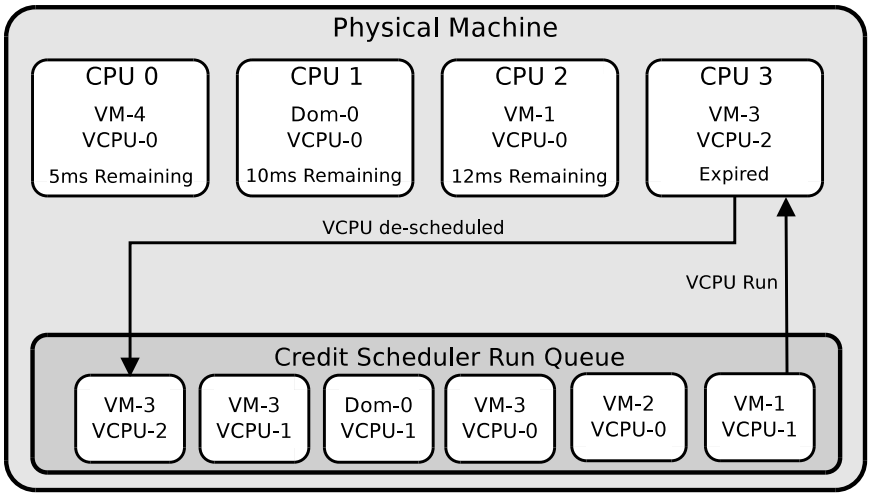


Figure 4.7: Xen Credit Scheduler
bottom

physical resources the scheduler will not assign them to a VM which has exceeded its utilization cap. The weighted priority is an integer assigned by the system administrator to each VM running on the Xen system. The weight represents a proportional share of the CPU, for example if virtual machine VM-1 has the default scheduler weight of 128 and VM-2 has a weight of 256, VM-2 will be scheduled twice as often on the physical CPUs. The utilization cap is provided as a percentage and corresponds to the amount of physical CPU time which can be consumed. For example, a cap of 50% corresponds to half of a physical CPU and a cap of 200% indicates the VM can fully occupy 2 physical CPUs. Figure 4.7 shows a simplified version of the Xen credit schedulers global load balancing. As can be seen we have a number of VCPUs belonging to different VMs running on our 4 physical CPUs. Each one is given a maximum time slice of 30 ms. Execution for these VCPUs end when the virtual machine consumes its time slice, blocks, or yields its remaining time. In the example in Figure 4.7 VM-3's Virtual CPU-2 consumes its entire time slice and is placed at the end of the run queue. VM-1 is at the front of the run queue so its VCPU is scheduled to run on CPU-3. It is important to note that a VM that is de-scheduled due to an expired time slice or blocking is not necessarily sent to the end of the run queue. For example, a VM with a high priority weight can be placed in front of lower priority VMs. Also, a VM which blocks or yields without consuming its time slice will be placed in front of VMs which have consumed their time slice.

4.2.3 Xen Network Architecture

The Xen network subsystem must pass packets through a number of steps and CPU process in order to deliver the packet to a VMs' network layer. Figure 4.8 shows a simplified drawing of the steps involved in delivering packets to a Xen VM. On receipt of a packet in Xen,

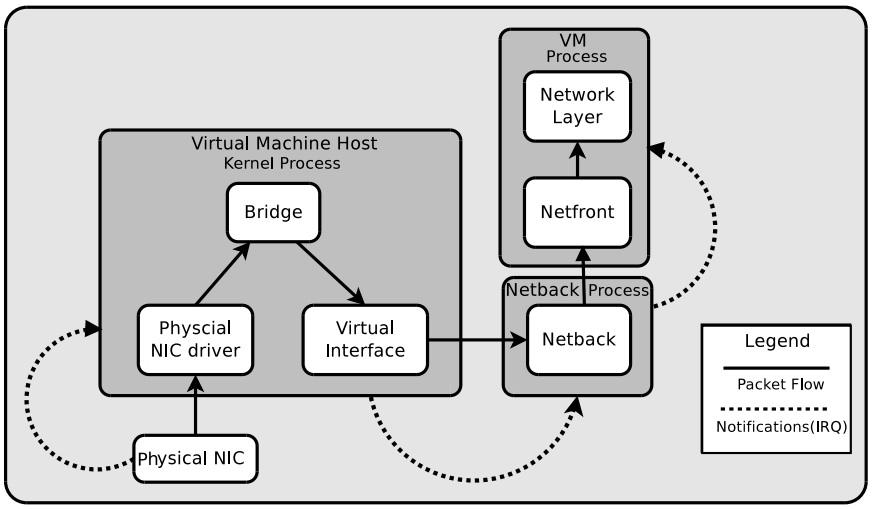
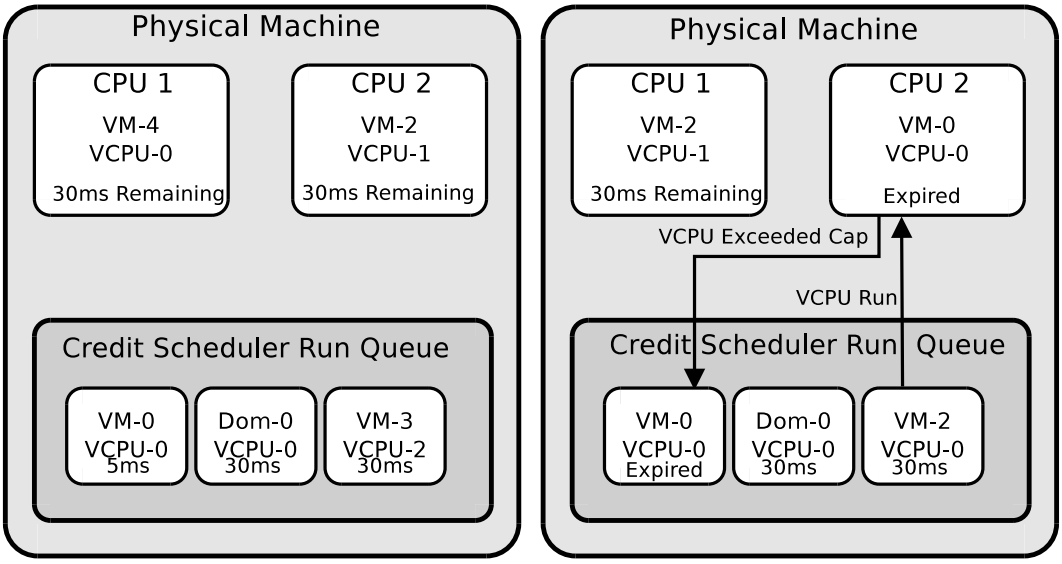


Figure 4.8: Xen Simplified Network Architecture
bottom

the physical NIC driver delivers the packet to the bridge; Once the packet arrives at the bridge it is transferred to the Netback module, which allocates buffers for the packet and notifies Netfront, in the Guest VM, of the incoming packet; Finally, Netfront receives the packet and passes it to the guests' network layer. As can be seen the packet must traverse several processes in multiple domains before it is finally delivered to the network layer in the virtual machine. The netback process for example runs in the driver domain (Domain-0) and is responsible for copying packets to and from the virtual machine's memory space. The netfront module runs inside the virtual machine, essentially operating as a network interface controller. Since these two modules run in different Domains there is potential that the credit scheduler can schedule these Domains far apart in time. For example, the netback process for a VM may be set to run at the front of the run queue and our VM is at the end of the queue because it consumed all its credits on a CPU task. Potentially this VM must wait 30ms now before it can process its packets. Next, we will discuss how the network architecture and CPU scheduler interact with each other to form the unstable performance experienced by our EC2 instances.

4.2.4 Reasons for Delay and Bandwidth Instability

We can roughly split the delay and bandwidth instability into two categories. The first category we define as "co-runner interference", which is the bandwidth and delay instability experienced when the VM's VCPUs are working in a near idle condition. For example, the VM's VCPU is only processing network traffic and not performing any computational intensive tasks. The second category we define as "self interference", which is bandwidth and delay instability experienced when the VMs VCPU are running both bandwidth and computational intensive tasks. We start our discussion with co-runner interference.



(a) Unstable Performance (near idle VCPU) (b) Unstable Performance (Saturated VCPU)

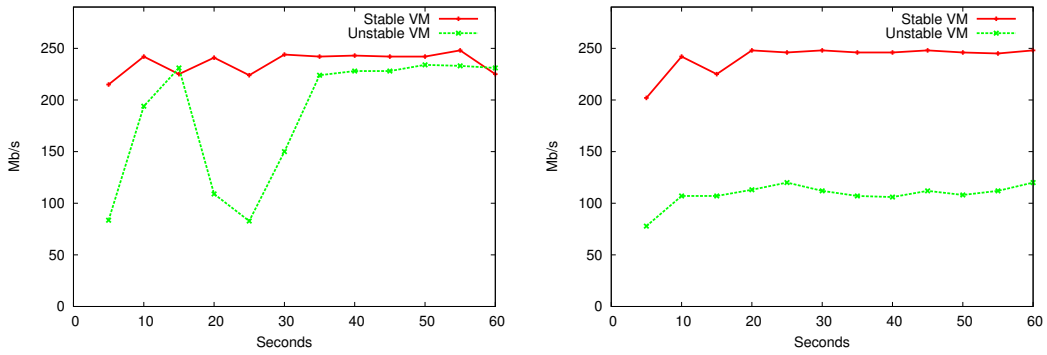
Figure 4.9: Unstable Performance
bottom

Co-runner interference was a major contribution factor to the unstable TCP performance experienced in Figure 4.1, by 8 out of 10 of our EC2 large instances while they were idle. This instability is created by the Xen scheduler scheduling bandwidth intensive and computational intensive virtual machines to the same physical processors. Since the bandwidth intensive tasks rarely use their whole time slice they often yield and are replaced by computational VMs, which always consume their entire 30ms time slice. Also, as discussed previously the netback process for the VM runs in the driver domain (Domain-0), which can also be blocked from running for long periods by these computational intensive tasks. If either Domain-0 or our bandwidth intensive VM have to wait a multiple of 30 ms between executions on the CPU, the TCP congestion algorithm will detect a congestion event, leading to a drop in throughput. Figure 4.9a shows an example where VM-0 is our bandwidth intensive VM, which has only 5ms CPU processing to perform when it is scheduled to run. However, all physical CPUs are currently occupied by VMs which will use their entire time slice of 30ms, and the next two VCPUs on the run queue also have 30ms of processing. In this example VM-0 will have to wait up to 60 ms before it gets a chance to run creating a significant delay in its packet processing. These delays in processing have very little effect on our idle large instances' UDP performance seen in Table 4.6, this is because UDP does not control for congestion so infrequent spikes in the RTT do not affect the bandwidth. Next we discuss bandwidth/delay instability due to self interference.

Since a VM currently has no way to indicate the priority of its tasks running on its VCPUs to the Xen hypervisor, a low priority CPU processing task can consume an entire 30ms time slice that may have actually been required for stable network performance. If

a VM expends its entire CPU time slice it will likely go to the end of the credit scheduler run queue and have to wait a considerable amount of time before it reaches the front of the queue again. Figure 4.9b shows a bandwidth intensive virtual machine, VM-0, which has used its entire CPU time slice on a low priority computational task. VM-0 is de-scheduled and placed at the end of the run queue, where on a busy physical machine it could be 30ms or more before VM-0 is scheduled again. If the Xen system administrator has decided to use the Xen credit scheduler CPU cap option on the VM-0 it will not be scheduled again until the next scheduler accounting period even if there are idle resources on the physical system. This is because the credit scheduler cap option enforces rigid execution rules on a VM and is not necessarily work conserving. Our previous EC2 experiments showed that self interference creates terrible performance for the eight unstable instances. Not only does the RTT increase dramatically in Figure 4.6 but the UDP experiments in Figure 4.3 make it clear that the data rate of these instances drop considerably for both sending and receiving. In the receiving UDP experiments we also see an increase in drop rate because the VM's netback thread must drop packets, likely because the VM is not emptying the Domain-0 receive buffers often enough. The UDP sending experiment sees no increase in drop-rate since it is likely the VMs sending buffers are now full inside the VM and it is not given enough CPU time to pass the data to the netback thread; because of this the VM's operating system slows the UDP sending rate of the instance.

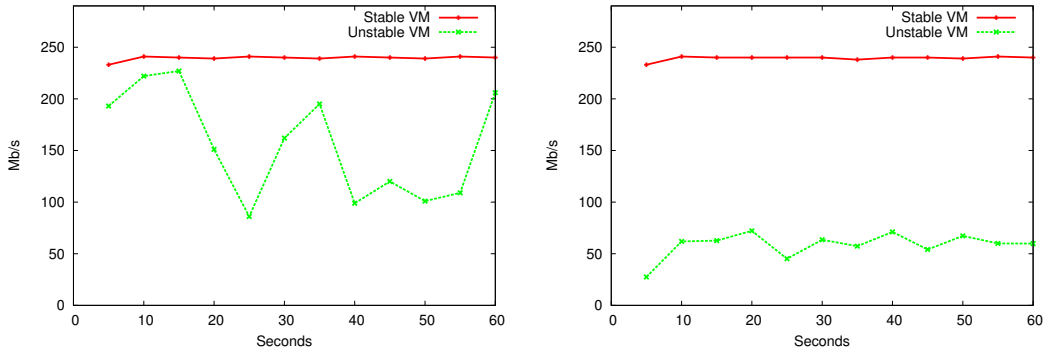
Finally, it is important to note that the two kinds of interference are not necessarily mutually exclusive. For example a Xen virtual machine with unstable idle performance will likely have even poorer performance when it saturates its VCPUs.



(a) TCP Sending Idle

(b) TCP Sending Load

Figure 4.10: Local Xen Instances Sending TCP Traffic
bottom



(a) TCP Receiving Idle

(b) TCP Receiving Load

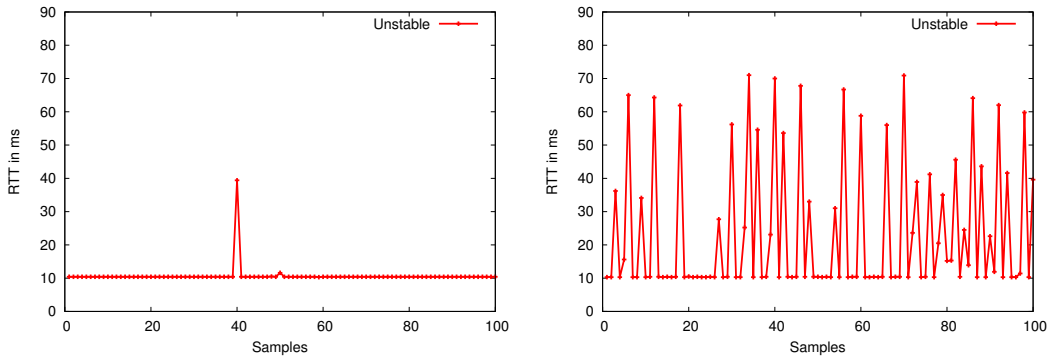
Figure 4.11: Local Xen Instances Receiving TCP Traffic
bottom

4.3 Verification Experiments

Our local cloud is a cloud platform based on Citrix's Cloud Stack management software. Each node has 6 CPU cores, 16 GB of main memory and 500 GB of local storage. Networking is provided through1000 Mb/s switches. The hypervisor used on our cloud nodes is Xen 4.1, which as of this writing is the latest stable version. The CPU scheduler algorithm used on our platform is the Xen credit scheduler. We create a cloud instance with similar memory and CPU specifications to an EC2 large instance. An EC2 large instance is configured with 7.5 GB main memory and 2 virtual CPUs with 4 EC2 compute worth of processing power. By comparing the performance of CPU benchmarks on EC2's cloud and our local cloud we have determined that 4 EC2 compute units is equivalent to 1.2 physical cores on our cloud platform. To recreate the network conditions experienced on our EC2 experiments we limit each VM to a network rate of 250 Mb/s. We also use the network emulator netem to emulate a 10 ms round trip time between our cloud instances and the remote host. Our remote host is a physical machine located one hop away from our platform and connected through 1000 Mb/s Ethernet. Like the EC2 experiments our VMs run Ubuntu 12.04 and any additional software packages are downloaded from the official repositories. We once again use iperf to test network performance and the sysbench CPU prime benchmark to create processor load.

In order to test unstable instances we create a situation similar to that seen in Figure 4.9a. We create two local VMs similar to EC2 large instances and two VMs similar to EC2 small instances. We assign these 4 VMs to three of our physical systems processor cores. Previously, we calculated each large instance required 1.2 processor cores and each small instance has 1/4th the processing power we required 0.3 cores per small VM. The total processor requirement for these 4 VMs is 3 processor cores. To ensure proper sharing of the resources we use the Xen credit scheduler optional cap feature to ensure each VM

uses only its proper proportion of CPU time. Next, we configure a single large VM and the two small VMs as computational intensive VMs by running sysbench CPU prime. The remaining large VM is our test VM on which we will test the network performance. To create a stable instance we simply provision a VM on one of our cloud servers, and pin the VM to physical processors which are not shared with others VMs.



(a) RTT idle (Unstable Instance)

(b) RTT load (Unstable Instance)

Figure 4.12: ICMP Round Trip Time Unstable Local Experiments bottom

4.3.1 TCP experiments

Similar to our EC2 experiments, we test the maximum TCP bandwidth of our VMs. We configure a single stable VM and a single unstable VM on our local cloud using the technique previously described. We begin with the sending experiments, which are given in Figure 4.10. As expected, the stable VM maintains a data rate of approximately 240 Mb/s both while the VM is idle and when we introduce a computational task using sysbench. The unstable VM, on the other hand, shows inconsistent performance as the TCP rate fluctuates rapidly both while the VMs vcpu is idle and while saturated. The unstable VMs idle TCP sending bandwidth is 186 Mb/s, which is almost 21% lower than our stable instance. When a computational task is introduced inside the unstable VM we see an even further drop in sending performance, which now only measures 108 Mb/s, far less than half the performance of the stable instance.

Next we look at the TCP receiving experiments shown in Figure 4.11. Once again our stable instance maintains consistent performance, receiving at a rate of approximately 240 Mb/s both while its VCPU is idle and while it is saturated. The unstable VM once again shows inconsistent performance both while idle and with a saturated VCPU. While idle the receiving performance of the unstable instance is only 156 Mb/s a drop of over 34% when compared to the stable instance. When the unstable's VCPU is saturated the performance drops by over 75% to only 59 Mb/s.

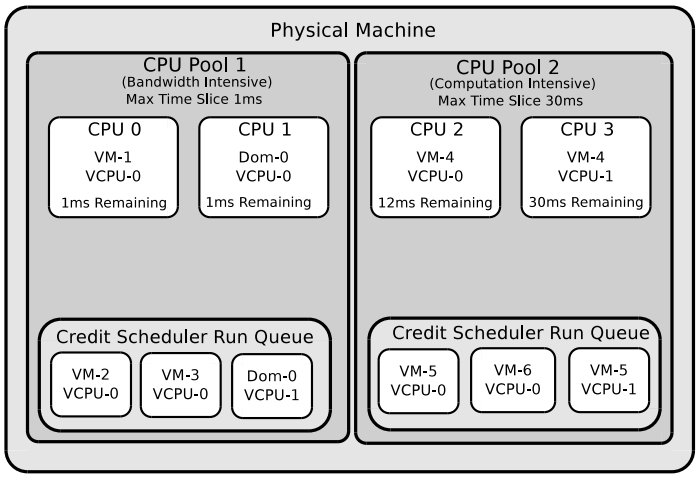


Figure 4.13: Multiple Pools for Different VM classes
bottom

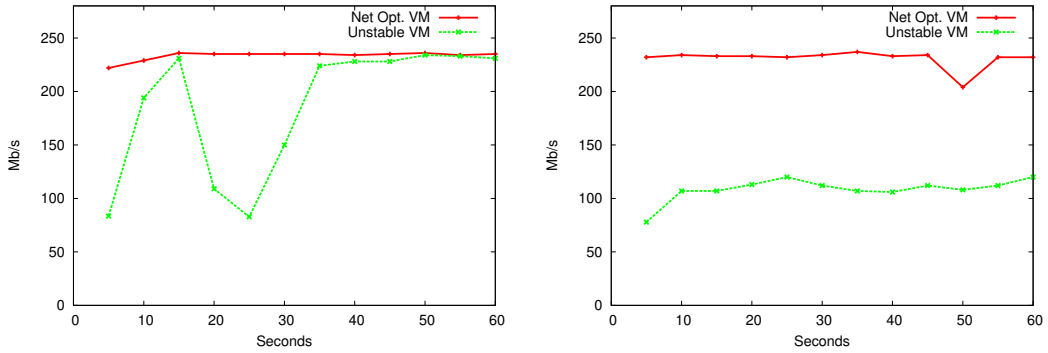
As can be seen in our experiments the TCP performance of our unstable local cloud instances are nearly identical to the TCP performance experienced by the EC2 instances in section 4.1. This near identical performance is a very strong indication that inconsistent TCP performance experienced by our EC2 instances is likely due to VCPU scheduling issues in the Xen hypervisor and not necessarily the network conditions in the data-center.

4.3.2 ICMP RTT experiments

As in the EC2 experiments we wanted to investigate the effect computational work has on the round trip time (RTT) performance of our instances. We used the hping3 tool to create ICMP echo request packets with 100 bytes of data each and sent them to our instances at a rate of 10 per second. As is to be expected the stable instance showed no increase in RTT while its VCPU is idle or when it is saturated with a computational task. Figure 4.12 gives the unstable instance performance. When the unstable instance's VCPUs are idle we see a single increase to above 30 ms. However, as can be seen introducing a load to the VM causes significant RTT variation. Similar to what was seen in our EC2 experiments these increases are generally in approximate multiples of 30 ms. This is expected as the delay is being caused by the unstable instance exhausting its 30 ms time slice on the CPU and having to wait at least 30 ms to get another chance to run.

4.4 Mitigate Performance Degradation and Instability

Since the presence of co-runner interference is not something that can be controlled from inside the VM, we must change the scheduler in-order to improve performance. Much work has been done on the Xen scheduler to increase the performance of latency sensitive applications. For example, work has been done to improve the response time of network based



(a) TCP Sending Idle

(b) TCP Sending Load

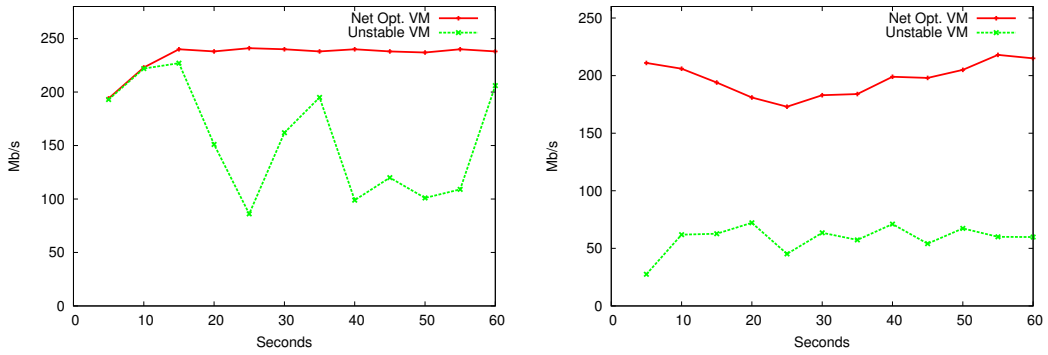
Figure 4.14: Sending TCP Traffic effect of 1 ms time slice pool

applications hosted in Xen virtual machines [21]. Also, researchers at George Washington University modified the Xen scheduler to prioritize interactive virtual machines, which were used to create virtual desktops [28]. However, these modifications often entail a tradeoff of less efficient computational performance. This lower computational performance is often due to a decrease in the maximum time slice assigned to a virtual machine. This shorter time slice means that VMs are scheduled more often but also results in higher number of processor context switches and lower processor cache efficiency, since VMs are being constantly interrupted while running. These negative side effects are likely one of the reasons why public clouds like EC2 appear to use the standard Xen credit scheduler.

To solve this impasse between the scheduling requirements of computational tasks and network intensive tasks, we propose running two different scheduling strategies in Xen at the same time, one which optimizes network performance and the other which optimizes computational tasks. The owner of the VM is free to choose which scheduling strategy (s)he wants the VM to use. Recently, the Xen maintainers have added the notion of CPU pools into the experimental version of the Xen Hypervisor. CPU pools allow us to pool physical CPUs under different configurations including scheduling rules.

To evaluate the effectiveness of this strategy we install the latest experimental version of Xen on one of our cloud platforms. We configure 2 CPU pools, Pool 1 and Pool 2, each with 3 physical cores. We configure Pool 1 to be our network optimized pool by selecting a maximum CPU time slice of 1 ms. CPU pool 2 is optimized for computational tasks so we select a maximum time slice of 30 ms. For both pools we use the standard Xen credit scheduler. Figure 4.13 shows the architecture of our Xen system. To recreate co-runner interference we once again create 4 VMs, 2 large VMs and 2 small VMs, and assign them to CPU pool 1. Once again, one large and two of the small VMs are configured to saturate their VCPUs with computational tasks. The remaining large VM is the machine we will

test the performance on. We also assign a single VM with 3 VCPUs to pool 2 to ensure the physical system is fully utilized.



(a) TCP Receiving Idle

(b) TCP Receiving Load

Figure 4.15: Receiving TCP Traffic effect of 1 ms time slice pool

The sending results for our network optimized CPU pool experiments are given in Figure 4.14 and the receiving results are given in Figure 4.15. As can be seen, the use of the 1ms credit maximum time slice in our network optimized pool has greatly improved TCP performance both while under load and while idle for sending and receiving. With identical conditions to our previous local experiments, a bandwidth optimized VM achieves a stable sending rate of 230 Mb/s while the VCPU is idle. Our unstable VM, in the same test, suffers from inconsistent performance and a data rate of only 185 Mb/s. When we introduce a computational load, our network optimized VM achieves near identical performance to its base line. This is a massive improvement as it is experiencing the same interference as the unstable VM but it is not exhibiting the same performance loss. However, it does exhibit a small amount of instability, dropping slightly due to self interference and recovering quickly. The unstable VM, on the other hand, exhibits a massive drop in performance, falling to under 110 Mb/s. In our TCP receiving test our network optimized VM stays stable at a throughput of nearly 235 Mb/s while idle. The unstable VM, on the other hand, performs inconsistently, achieving a throughput of only 155 Mb/s. When a computational load is introduced both our network optimized VM and the unstable VM are unable to maintain their baseline performance. However, our network optimized VM manages a performance of nearly 87% of the baseline, receiving at a rate of approximately 200 Mb/s. From our experiments it is clear that using multiple CPU pools, some with network optimizations such as shorter VM CPU time slices, can greatly reduce co-runner interference experienced by virtual machines. Our experiments also show that the use of the 1 ms time-slice effects the running time of our CPU tasks by less than 8%. However, the use of bandwidth optimized CPU pools does not solve all issues, as the network receiving performance does fall when a computational task is run alongside the network intensive task. It is likely combining our

1 ms CPU pool fix with a TCP acknowledgement offloading technique like the previously discussed vSnoop could improve the performance even further [36].

4.5 Discussion

In this chapter we systematically investigated the effect virtualization overhead has on network performance in public cloud computing. We discover that in public cloud computing platforms such as Amazon EC2, different VMs can suffer from different amounts of network performance instability. For example, some VMs show no signs of instability while others show extreme network performance instability. Through system analysis and local experiments we have discovered that this instability is often caused by computational load on the physical systems and can be roughly split into two types, self interference and co-runner interference. We then implement practical modifications in existing software that show that this performance instability can be mitigated to a large degree.

From our observations in undertaking this set of experiments, we noted that some Amazon cloud datacentre zones have a higher percentage of instances with unstable network performance than others. For example, the majority of EC2 large instances provisioned in Amazon's Oregon datacentre Zone A showed very unstable performance. However, this datacentre recently added a new zone, Zone C; when we allocated our large instances in this zone, a smaller proportion showed unstable performance. This is likely because the new zone has fewer users, and thus currently has a lower utilization. Given time, more users will allocate resources in this zone, and this will lead to an increase in network instability.

Future research in this area should focus on the best route for deploying mitigation strategies in large scale datacentres. Our results are encouraging that network performance and stability can be greatly improved with practical modifications to existing systems.

Chapter 5

Virtualized GPU Performance For Cloud Gaming Applications

Fueled by elastic resource provisioning, reduced costs and unparalleled scalability, *cloud computing* is drastically changing the operation and business models of the IT industry [2]. A wide range of conventional applications, from file sharing and document synchronization to media streaming, have experienced a great leap forward in terms of system efficiency and usability through leveraging cloud resources with *computation offloading*. Recently, advances in cloud technology have expanded to facilitate offloading more complex tasks such as high definition 3D rendering, which turns the idea of *Cloud Gaming* into a reality. Cloud gaming, in its simplest form, renders an interactive gaming application remotely in the cloud and streams the scenes as a video sequence back to the player over the Internet. A cloud gaming player interacts with the application through a *thin client*, which is responsible for displaying the video from the cloud rendering server as well as collecting the player's commands and sending the interactions back to the cloud.

This new paradigm of gaming services brings immense benefits by expanding the user base to the vast number of less-powerful devices that support thin clients only, particularly smartphones and tablets [9]. Extensive studies have explored the potential of the cloud for gaming and addressed challenges therein [58] [12] [41]. Open-source cloud gaming systems such as *GamingAnywhere* for Android OS [26] have been developed. We have also seen industrial deployment, e.g., OnLive [47] and Gaikai [19].

These existing cloud gaming platforms tend to focus on private, non-virtualized environments with proprietary hardware, where each user is mapped in a one-to-one fashion to a physical machine in the cloud. Modern public cloud platforms heavily rely on *virtualization*, which allows multiple *virtual machines* to share the underlying physical resources, making truly-scalable *play-as-you-go* service possible. Despite the simplicity and ease of deployment, existing cloud gaming platforms have yet to unleash the full potentials of the modern cloud towards the expansion to ultra-large scale with flexible services.

Migrating gaming to a public cloud, e.g., Amazon EC2, is non-trivial however. The system modules should be carefully planned for effective virtual resource sharing with minimum overhead. Moreover, as the complexity of 3D rendering increases, modern game engines not only rely on the general purpose CPU for computation, but also on dedicated *graphical processing units* (GPUs). While GPU cards have been virtualized to some degree in modern virtualization systems, their performance has historically been poor given the ultra-high memory transfer demand and the unique data flows [15]. Recent advances in terms of both hardware and software design have not only increased the usability and performance of GPUs, but created new classes of GPUs specifically for virtualized environments. A representative is NVIDIA's recently released GRID Class GPUs, which allows multiple virtualized systems to each utilize a dedicated GPU, by placing several logical GPUs on the same physical GPU board. It also contains a hardware H.264 video encoder, and similar onboard hardware encoders are available in GPUs from other industry leaders such as Intel's Quick Sync Video (QSV) and AMD's Video Coding Engine (VCE). These new hardware advances from nearly every major GPU vendor allow us to take a step towards in the deployment of online gaming systems in a public cloud environment.

5.1 Advanced GPU Pass-through and Gaming Performance

To understand the performance improvement and whether the technology is ready for gaming over public cloud, we have conducted a series of experiments over both an earlier GPU pass-through platform and the latest platform. With direct access to the hardware, these local virtualized platforms facilitate the measurement of virtualization overhead on game applications. We are also able to measure energy consumption of our local platform, which we are not able to obtain from a public cloud platform. As such, our tests covers a broad spectrum of metrics, including frame rate, energy consumption, and CPU memory bandwidth, both with a single user exclusively using the entire resources and with multiple users sharing resources. We will now compare an older more primitive implementation of virtualized device pass-through from 2011 to a newer more optimized version from 2014. For brevity, we will refer to these two platforms as "earlier" and "advanced", respectively. Since we are interested in the impact virtualization overhead has on gaming performance, we also compare these systems to their optimal non-virtualized performance(referred to as "bare-metal" performance).

5.1.1 Earlier GPU pass-through Platform (2011)

Our first test system is a server with an AMD Phenom II 1045t 6-core processor running at 2.7 Ghz. The motherboard's chipset is based on AMD's 990X, and we enabled AMD-V and AMD-Vi in the BIOS as they are required for Hardware Virtual Machines (HVM) support and device pass-through. The server is equipped with 16 GB of 1333 MHz DDR-3 SDRAM

	Bare-metal	Virtual Machine	Energy
Doom 3 (Joules) Old	3696.9 J	11722.3 J	217%
Unigine Sanctuary (Watts)	265.2 W	273.6 W	3.2%
Doom 3 (Joules) Adv.	1593.9 J	1676.7 J	5.2%
Unigine Sanctuary (Watts)	289.2 W	301.7 W	4.4%

Table 5.1: Power Consumption: Games Bare-metal vs. Virtual Machine

and the physical network interface is a 1000 Mb/s Broadcom Ethernet adapter attached to the PCI-E bus. The GPU is an AMD-based Sapphire HD 5830 Xtreme with 1 GB of GDDR5 memory.

The Xen 4.0 hypervisor is installed on our test system and the host and VM guests used Debian as their operating system. We configure Xen to use the HVM mode, since the GPU pass-through requires hardware virtualization extensions. The VM is given access to 6 VCPUs and 8048 MB of RAM.

5.1.2 Advanced GPU pass-through Platform (2014)

Our second system is a state-of-the-art server with an Intel Haswell Xeon E3-1245 quad core (8 threads) processor. The motherboard utilizes Intel's C226 chipset, which is one of Intel's latest server chipsets, supporting device pass-through using VT-D. The server has 16 GB of 1600 MHz ECC DDR-3 memory installed. Networking is provided through an Intel i217LM 1000 Mb/s Ethernet card. We have also installed an AMD based Sapphire R9-280x GPU with 3 GB of GDDR5 memory, which is representative of advanced GPUs in the market. The Xen 4.1 hypervisor is installed and the VM guests again use Debian as their operating system. On this basis, we configure Xen to use the HVM mode, and the VM is given access to 8 VCPUs and 8048 MB of RAM. Since our system's physical hardware supports Intel's hyper-threading technology we enable it in the BIOS for both the virtualized and bare-metal experiments.

5.2 Comparison and Benchmarks

As the optimal baseline for comparison, for both systems we run each test on a bare-metal setup with no virtualization, i.e., the system has direct access to the hardware. The same drivers, packages and kernel were used as in the previous setup. This particular configuration enabled us to calculate the amount of performance degradation that a virtualized system can experience.

To determine the overall system's power consumption, referred to as *wall power*, we have wired a digital multi-meter (Mastech MAS-345) into the AC input power line of our system. The meter has an accuracy rating of ± 1.5%. We read the data from our meter

using a data logger PCLink installed on a workstation, and collect samples every second throughout our experiments. Both systems are powered by 750 Watt 80 Plus gold power supplies with active power factor correction.

To compare the pass-through performance, we have selected two game engines, both of which have cross-platform implementation, and can run natively on our Debian Linux machines. The first is `Doom 3`, which is a popular game released in 2005, and utilizes OpenGL to provide high quality graphics. The second is the Unigine's `Sanctuary` benchmark[1], which is an advanced benchmarking tool that runs on both Windows and Linux. The Unigine engine uses the latest OpenGL hardware to provide rich graphics that are critical to many state-of-the-art games. For each of the following experiments, we run each benchmark three times and depict the average. For Doom3 and Sanctuary, we give the results in *frames per second* (fps).

5.2.1 Frame Rate and Energy Consumption

For Doom 3, we use the ultra high graphics settings with 8x Anti-Aliasing (AA) and 8x Anisotropic Filtering (AF), with a display resolution of 1920 x 1080 pixels (1080p). To perform the actual benchmark, we utilize Doom 3's built in time demo, which loads a predefined sequence of game frames as fast as the system will render them.

In Figure 5.1, we show the results of frame rates, running on our bare metal systems as well as on the Xen virtualized systems. We start the discussion with our older 2011 server. This bare metal system performs at 126.2 fps, while our virtualized system dramatically falls over 65% to 39.2 fps. Our newer 2014 system processing the frames at over 274 fps when running directly on the hardware and falling less than 3% when run inside a virtual machine.

Table 5.1 gives the energy consumption experienced by our machines in this experiment, measured in joules (since each system has a different running time in the test). The older system consumes over twice the amount of energy to perform the Doom 3 time demo when the system is virtualized. The advanced platform fares much better with virtualization, only adding 5.4% to energy consumption. This first virtualization experiment makes it clear that the device pass-through technology has come a long way in terms of performance and overhead. The advanced platform performs within 3% of the optimal bare-metal result consuming only 5% more energy. The energy efficiency of this advanced platform is quite impressive, considering that a virtualized system must maintain its host's software hypervisor as well as many hardware devices such as the IOMMU, shadow TLB and the CPUs virtualization extensions.

To confirm the performance implications with newer and more advanced OpenGL implementations, we next run the Unigine Sanctuary benchmark at 1920 x 1080, with all

[1] `https://unigine.com/products/sanctuary/`

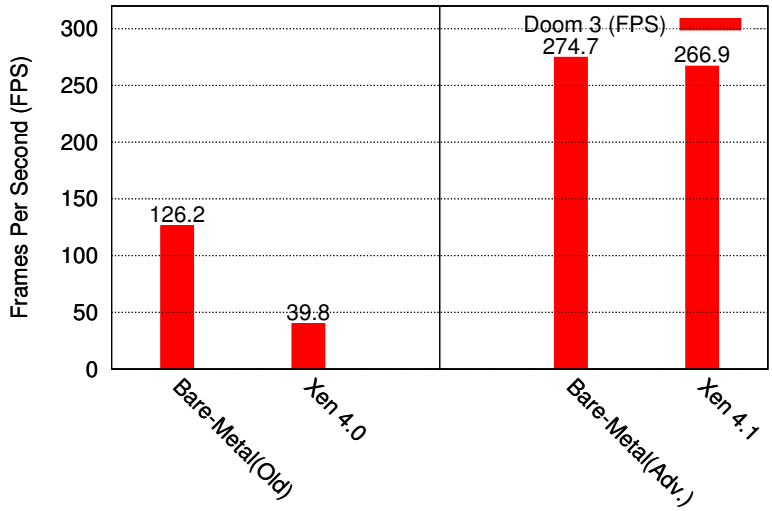


Figure 5.1: Doom 3 Performance

high-detail settings enabled. To further stress the GPUs, we enable 8x Anti-Aliasing (AA) and 16x Anisotropic Filtering (AF). We run the benchmark mode 3 times and measure the average frame rates and the energy consumption of the system. In this experiment, the running time is consistent between each run and platform; we express the results in watts (Joules per Second). The results are given in Figure 5.2. Once again, we see that our earlier virtualized system shows significant signs of performance degradation when compared to its bare-metal optimal. The earlier system drops from 84.4 fps to 51 fps when virtualized, i.e., nearly 40%. The advanced system has near identical performance when the game engine is running in a virtualized environment or directly on the hardware. Table 5.1 further shows that our advanced system uses a measurable more power but still impressively remains at an increase of only 4.4% more power when virtualized. The power consumption of the earlier system appears to have only increased by 3.2%; on the surface this looks like a positive trait, but indeed comes with the cost of nearly 40% lower video frame rate.

5.2.2 GPU Memory Bandwidth

Memory transfer from the system's main memory to the GPU can be a bottleneck for gaming, which can be even more severe when the transfer is performed in a virtualized environment [60]. This would affect the game's performance as well, because both Doom 3 and the Unigine engine must constantly move data from the main memory to the GPU for processing. To understand the impact, we ran a simple memory bandwidth experiment written in OpenCL by NVIDIA[2]. We tested three different copy operations, from host's main memory (DDR3) to the GPU device's global memory (GDDR5), from the device's

[2]Code is available in the NVIDIA OCL SDK at: `http://developer.NVIDIA.com/opencl`

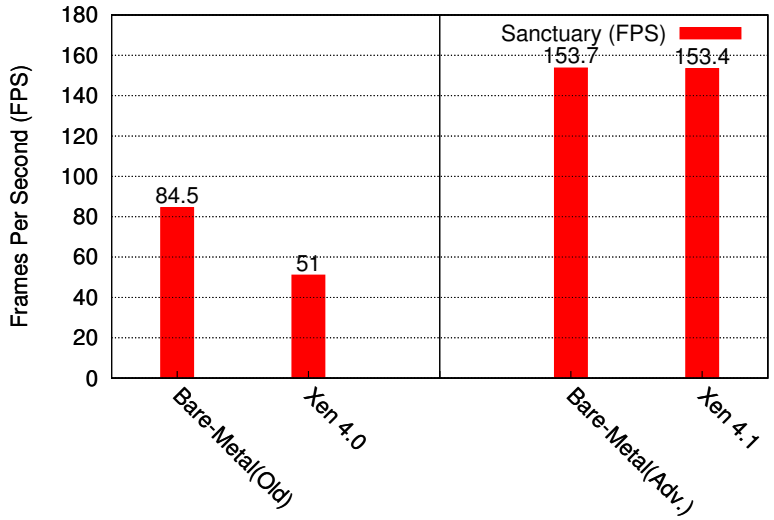


Figure 5.2: Unigine Sanctuary Performance

global memory to the host's main memory and finally from the devices global memory to another location on the devices global memory. We give the results for host-to-device and device-to-host experiments in Figure 5.3.

For both systems, the bare-metal outperforms the VMs in terms of memory bandwidth across the PCI-E bus to the GPU. The earlier system loses over 40% of its memory transfer to and from the host memory when compared to the bare-metal system. The newer more advanced platform degrades even more, losing over 60% of its memory transfer speed when virtualized. Interestingly, all virtualized systems sustain a nearly identical performance to their bare-metal counterpart for device-to-device copy: the earlier platform at 66,400 MB/s while the more advanced platform at 194,800 MB/s. This indicates that, once the data is transferred from the virtual machine to the GPU, the commands that operate exclusively on the GPU are much less susceptible to the overhead incurred by the virtualization system.

Our results indicate that, even though the gaming performance issues have largely disappeared in the more modern advanced platform, the memory transfer from main memory across the PCI-E bus remains a severe bottleneck. It can become an issue during the gaming industry's transition from 1080p to the newer high memory requirements of 4k (4096 x 2160) UHD resolution.

5.3 Advanced Device Pass-through Experiments

The previous experiments have shown that running gaming applications inside a virtual machine is now feasible with little overhead. The observation however is confined to virtual machines having access to the entire system. In a real-world deployment over a public cloud,

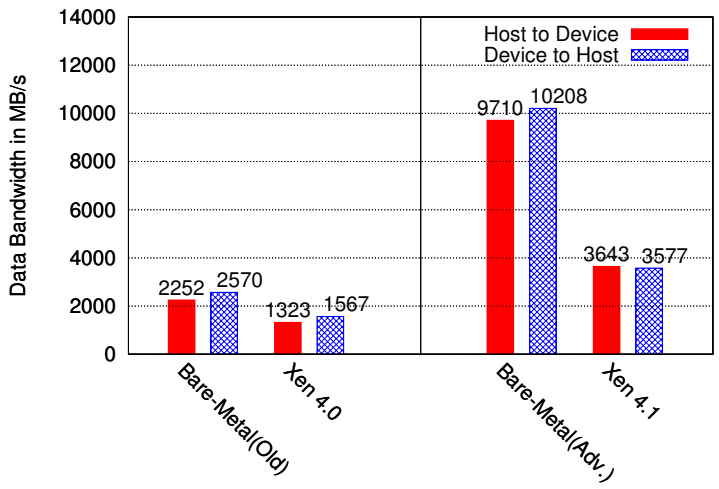


Figure 5.3: Memory Bandwidth by System

we are more interested in sharing a physical machine and its resources among multiple users. To this end, we upgraded our advanced platform to three R9 280x GPUs, to test the performance and energy implication of multiple gaming applications running at one time on our platform. We required three GPUs, as we tested the performance of direct device pass-through, which is a one-to-one mapping between GPU and VM. To test the performance of other widely deployed 3D rendering languages, such as Microsoft's DirectX, we further created a Windows server 2008 virtual machine and corresponding bare-metal installation on our platform.

Our first advanced experiment utilizes Futuremark's widely deployed gaming benchmark `3DMark`[3]. Specifically, we use the `Fire Strike` module, which is a 1080p benchmark utilizing the latest DirectX features (DX11) that stresses not only the 3D rendering capabilities of the system but also the physics calculations done on the CPU. We run the entire Fire Strike benchmark three times and provide the average graphics and physics frame rates. Once again, we collect the total energy consumption experienced by the system running the benchmark. We first run the benchmark on our bare-metal windows server to determine the optimal baseline. We then perform experiments using virtual machines, assigning each VM a dedicated R9 280x GPU and 4 GB of RAM. We test the performance of different numbers of VMs ranging from one to three concurrently running the 3DMark benchmark.

The performance results for the 3DMark experiment are given in Figure 5.4. We first look at the average frame rate; our bare-metal system achieves 34.71 fps and our single VM experiment achieves 34.69 fps. They are within 0.1% of each other after 3 runs, that is they are statistically identical. The 2-VM case drops to an average of 34.55 fps, which is still only a small drop of about 0.4%. Finally, with 3 VMs performing GPU-intensive operations,

[3] `http://www.3dmark.com/`

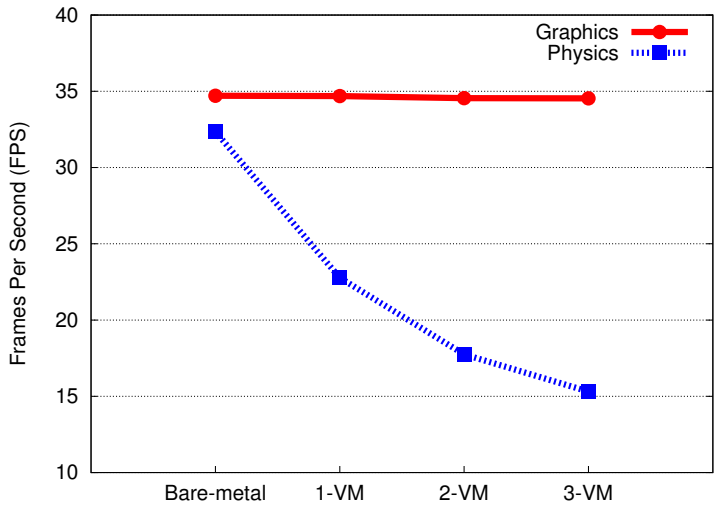


Figure 5.4: Advanced: 3DMark Frame Rate

the average score drops to 34.54 fps, which is still less than 0.5% drop in performance. In terms of graphic intensive gaming applications, very little performance is lost when adding more concurrent VMs. Also presented in Figure 5.4 is the 3DMark's CPU intensive physics gaming benchmark. Immediately we see a large difference between the bare-metal and the single virtual machine performance, even though they have access to the same amount of CPU resources. Our virtual system suffers 30% performance degradation from its optimal bare-metal performance. We have found that this is because the virtualized system does not properly use the hyper-threading provided by the Intel CPU. To show this, we disable the hyper-threading in our system's BIOS and re-run Fire-Strike on the bare-metal platform, which results in a physics test frame rate drop from 32.35 fps to 23.04 fps, making the performance of the bare-metal and the virtual machine within 1% of each other.

Hyper-threading allows multiple software threads to be loaded onto a physical core at one time. Thus, hyper-threading can potentially improve the performance of multi-threaded applications. However, the performance gain is application-dependent; if functioning fully, it can offer improved CPU utilization. Although in many cases game performance is bounded by the GPU, those games that introduce highly intensive CPU workloads and are properly optimized for hyperthreading may still benefit, and the 3DMark's physics benchmark in our experiment is clearly such a case. Our results suggest that certain gaming related optimizations could be applied to the Xen hypervisor to allow it to fully utilize hyper-threading for gaming applications.

5.3.1 3DMark Windows and DirectX 11 Performance

Next we run two VMs concurrently, and we find that the physics performance drops to just over 55% of the bare-metal base-line, and running 3 VMs drops to under 50% of the

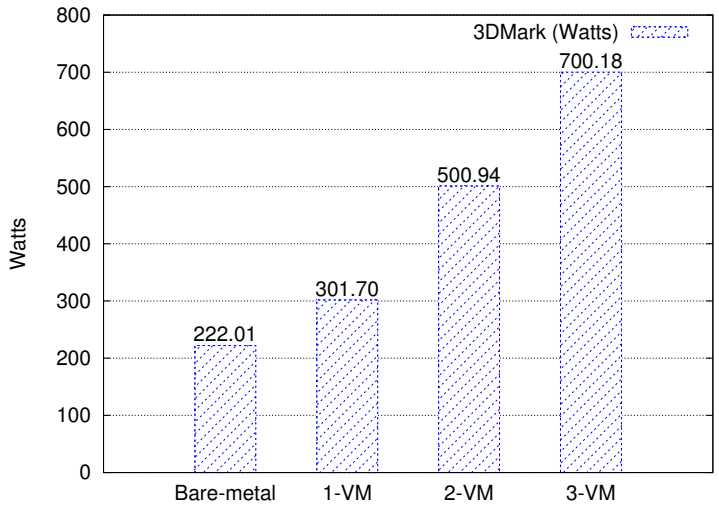


Figure 5.5: Advanced: 3DMark Power Usage

# of Instances	Bare-metal (fps)	VM (fps)
1	29.9	29.2
2	30.3	28.9
3	29.8	28.9

Table 5.2: Advanced: Multiple Instances Average Frame Rate

bare-metal base-line. Although the single VM experiment is surprising, the performance loss in the 2- and 3-VM test are more expected. Unlike the GPU intensive graphics modules in this benchmark, the CPU intensive physics modules are all processed by a single physical component, the CPU. Due to this device sharing, multiple VMs competing for the same CPU would affect each others' performance. The 3DMark's physics benchmark has been specifically programmed to use all the available CPUs. Other than the frame rate, this benchmark also gives the score as a form of general evaluation, and fortunately, a large public accessible dataset is available[4], which can be used to determine how a particular system performs when compared to thousands of others. When searching this database, we find that, under the 3 VM case, with a score of approximately 5000, each machine is achieving the gaming performance of a midrange Intel Core i5-2300 processor. For many gaming applications, however, the GPU is the bottleneck, and thus a shared CPU may not impact the game performance.

The power consumption for this test is given in Figure 5.5, which shows that our bare-metal system uses approximately 35% less energy than the same application running inside the VM. Further, the power consumption increases greatly as we increase the number of concurrently running VMs. This is because each additional VM saturates another dedicated

[4] `http://www.3dmark.com/search`

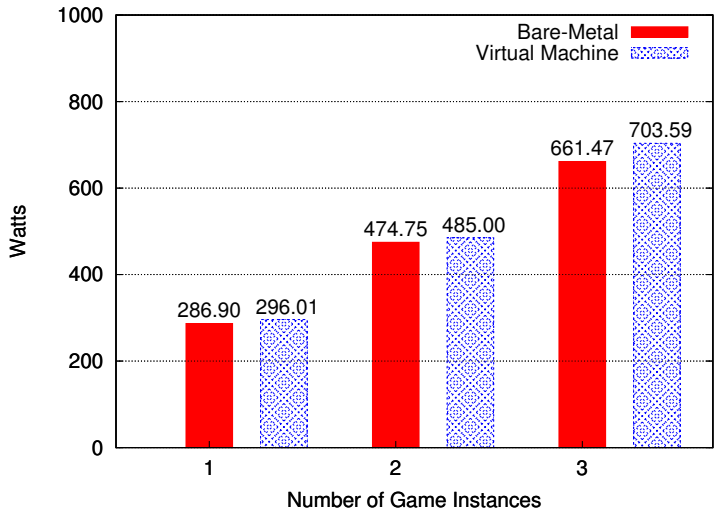


Figure 5.6: Advanced: Heaven Power Usage

physical GPU. Unfortunately, 3DMark only allows our bare-metal platform to run one instance of the Fire-Strike benchmark at a time, and hence we could not directly compare multiple instances of the same gaming application running directly on the hardware to those running inside different VMs.

5.3.2 Unigine Heaven - Multiple Instance Performance

To determine such difference in performance and energy consumption, our final experiment leverages the Unigine's latest benchmark Heaven[5], which supports the latest graphics rendering techniques such as DirectX 11 and OpenGL 4.0. Once again, we use a 1920 x 1080 resolution, with all high detail settings enabled and also utilize 8x Anti-Aliasing (AA) and 16x Anisotropic Filtering (AF).

The results in Table 5.2 suggest that the performance remains consistent as we add more instances of the application to either the bare-metal or virtualized system. For the bare-metal system, the average performance is approximately 30 regardless of the number of running game instances. The virtualized performance is approximately 5% lower at around 29 fps. Figure 5.6 further gives a comparison of the energy consumption. Although the energy consumption is higher in the virtual machine deployment, at its worst it is only about 6.5% more. This is impressive considering that in the virtual machine deployment, we actually have one operating system for the virtual machine host and one for each guest VM. Overall, given such important benefits of virtualization as performance isolation and resource sharing, the overhead of around $5 - 6.5\%$ for advanced virtualization is quite justifiable.

[5] `https://unigine.com/products/heaven/`

5.4 Discussion

Cloud gaming has attracted significant interest from both academia and industry, and its potentials and challenges have yet to be fully realized, particularly with the latest hardware and virtualization technology advances. This chapter presented an in-depth study in this direction. We closely examined the performance of modern virtualization systems equipped with virtualized GPU and pass-through techniques. Our results showed virtualization for GPU has greatly improved and is ready for gaming over a public cloud. A modern platform can host three concurrent gaming sessions running inside different VMs at 95% of the optimal performance of a bare-metal counterpart. Although there is degradation of memory transfer between a virtualized system's main memory and its assigned GPU, the game performance at the full HD resolution of 1080p was only marginally impacted.

Chapter 6

Concluding Remarks and Future Directions

In this thesis we systematically study critical aspects of computer virtualization technology. From security issues, such as Denial of Service (DoS) to energy consumption of virtualized network transactions, a proper understanding of the benefits and impacts of virtualization is critical to building higher performance systems. Using the knowledge found in this thesis and other research works the construction of more versatile and efficient architectures can be achieved.

6.1 Summary of this Thesis

First, we provided a study on the performance of modern virtualization solutions under DoS attacks. We experiment with the full spectrum of modern virtualization techniques, from paravirtualization, hardware virtualization, to container virtualization, with a comprehensive set of benchmarks. Our results reveal severe vulnerability of modern virtualization: even with relatively light attacks, the file system and memory access performance of VMs degrades at a much higher rate than their non-virtualized counterparts, and this is particularly true for hypervisor-based solutions. We further examine the root causes, with the goal of enhancing the robustness and security of these virtualization systems. Inspired by the findings, we implement a practical modification to the VirtIO drivers in the Linux KVM package, which effectively mitigates the overhead of a DoS attack by up to 40%.

Second, we present an empirical study on the power consumption of typical virtualization packages while performing network tasks. We find that both Hardware Virtualization and Paravirtualization add considerable energy overhead, affecting both sending and receiving, and a busy virtualized web-server may consume 40% more energy than its non-virtualized counterparts. Our detailed profiling on packet path reveals that a VM can take 5 times more cycles to deliver a packet than a bare-metal machine, and is also much less efficient on

caching. Without fundamental changes to the hypervisor-based VM architecture, we show that the use of adaptive packet buffering potentially reduces the overhead. Its practicality and effectiveness in power saving are validated through driver-level implementation and experiments.

Third, we find that such unstable network performance and variation phenomena can be prevalent and significant in the public cloud with both TCP and UDP traffic even with a lightly utilized network. We systematically investigate the impact and the root cause of this virtualization overhead in the cloud. Our in-depth measurement and detailed system analysis reveal that the performance variation and degradation are mainly due to the requirements of the CPU in both computation and network communication.

Finally, we show that modern public cloud platforms, which heavily rely on *virtualization* for efficient resource sharing, are now capable of providing computational resources needed to host 3D Gaming applications. This research shows that public cloud platforms such as Amazon's EC2 are now able to host high-performance gaming applications.

6.2 Future Directions

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

Virtualiztion has been a powerful concept and tool, however there are still vast improvements that can be made. One such advancement will come in the form of augmentation of the virtualized environments with new processing capabilities. One such avenue that needs further exploration is the development of GPGPU (General-Purpose computing on Graphics Processing Units) and FPGA (Field Programmable Gate Array) in Virtualized Cloud Computing environments. The wide deployment of these specialized components will greatly expand the potential of todays cloud offerings, not only from a user's perspective but also from that of a cloud provider. Users will be able to improve their data processing applications, for example accelerating map reduce by utilizing a GPU. Cloud providers can reduce their service hosting overhead by using these specialized devices to perform important infrastructure critical tasks such as processing packets in the virtual network. However, the presence of multiple processing engines such as the CPU, GPU, and FPGA presents an interesting problem for both the cloud user and cloud provider. The mapping of workloads to these devices to maximize the performance of the cloud becomes paramount. Equally important, how to streamline these tasks to result in lower energy consumption and heat production of the physical cloud systems. These are important questions I intend to answer

in my future research by studying both the theoretical aspects of these complex systems, as well as utilizing real world tests beds and measurements devices.

Bibliography

[1] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. Iommu: Strategies for mitigating the iotlb bottleneck. In *Computer Architecture*, pages 256–274. Springer, 2012.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] Ayan Banerjee, Tridib Mukherjee, Georgios Varsamopoulos, and Sandeep KS Gupta. Cooling-aware and thermal-aware workload placement for green hpc data centers. In *Green Computing Conference, 2010 International*, pages 245–256. IEEE, 2010.

[4] Cullen Bash and George Forman. Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–6. USENIX Association, 2007.

[5] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831. IEEE Computer Society, 2010.

[6] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *The Computer Journal*, 53(7):1045–1051, 2010.

[7] D.J. Bernstein. Syn cookies, 1996.

[8] Rajkumar Buyya, Anton Beloglazov, and Jemal Abawajy. Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges. *arXiv preprint arXiv:1006.0308*, 2010.

[9] W. Cai, M. Chen, and V. Leung. Toward gaming as a service. *IEEE Internet Computing*, 18(3):12–18, 2014.

[10] CDW. Cdw's server virtualization life cycle report, january 2010.

[11] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *Performance Evaluation Review*, 35(2):42, 2007.

[12] M. Claypool and D. Finkel. On the performance of onlive thin client games. *Springer Multimedia Systems Journal, Special Issue on Network Systems Support for Games*, PP(9):1–14, 2014.

[13] R.J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[14] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[15] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[16] W.M. Eddy. Defenses against tcp syn flooding attacks. *Cisco Internet Protocol Journal*, 8(4):2–16, 2006.

[17] W.M. Eddy. Tcp syn flooding attacks and common mitigations. 2007.

[18] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. *Cloud Computing*, pages 20–38, 2010.

[19] Gaikai. http://www.gaikai.com/.

[20] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. *ACM SIGARCH Computer Architecture News*, 40(1):411–422, 2012.

[21] S. Govindan, A.R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136. ACM, 2007.

[22] Mohamed Hefeeda and Cheng-Hsin Hsu. On burst transmission scheduling in mobile tv broadcast networks. *IEEE/ACM Transactions on Networking (TON)*, 18(2):610–623, 2010.

[23] M. Hemmati, A. Javadtalab, A. A. N. Shirehjini, S. Shirmohammadi, and T. Atici. Game as video: Bit rate reduction through adaptive object encoding. *in Proceedings of ACM NOSSDAV*, 2013.

[24] Mahdi Hemmati, Abbas Javadtalab, Ali Asghar Nazari Shirehjini, Shervin Shirmohammadi, and Tarik Arici. Game As Video: Bit Rate Reduction Through Adaptive Object Encoding. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '13, pages 7–12, New York, NY, USA, 2013. ACM.

[25] Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Placing Virtual Machines to Optimize Cloud Gaming Experience. *IEEE Transactions on Cloud Computing*, pages 1–1, 2014.

[26] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: An open cloud gaming system. In *Proceedings of the 4th ACM Multimedia Systems Conference*, MMSys '13, pages 36–47, New York, NY, USA, 2013. ACM.

[27] Qiang Huang, Fengqian Gao, Rui Wang, and Zhengwei Qi. Power consumption of virtual machine live migration in clouds. In *Communications and Mobile Computing (CMC), 2011 Third International Conference on*, pages 122–125. IEEE, 2011.

[28] J. Hwang and T. Wood. Adaptive dynamic priority scheduling for virtual desktop infrastructures. In *Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service*, page 16. IEEE Press, 2012.

[29] Intel. Intel xeon processor e5 family: Spec update.

[30] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, 2011.

[31] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 159–168, 2010.

[32] Jae-Wan Jang, Euiseong Seo, Heeseung Jo, and Jin-Soo Kim. A low-overhead networking mechanism for virtualized high-performance computing systems. *The Journal of Supercomputing*, pages 1–26. 10.1007/s11227-010-0444-9.

[33] D. Jayasinghe, S. Malkowski, Q. Wang, J. Li, P. Xiong, and C. Pu. Variations in performance and scalability when migrating n-tier applications to different clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 73–80. IEEE, 2011.

[34] C. Jin, H. Wang, and K.G. Shin. Hop-count filtering: an effective defense against spoofed ddos traffic. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 30–41. ACM, 2003.

[35] Yichao Jin, Yonggang Wen, and Qinghua Chen. Energy efficiency and server virtualization in data centers: An empirical investigation. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 133–138. IEEE, 2012.

[36] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[37] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 39–50. ACM, 2010.

[38] Sung-soo Kim, Kyoung-ill Kim, and Jongho Won. Multi-view Rendering Approach for Cloud-based Gaming Services. In *AFIN 2011, The Third International Conference on Advances in Future Internet*, pages 102–107, 2011.

[39] Bhavani Krishnan, Hrishikesh Amur, Ada Gavrilovska, and Karsten Schwan. Vm power metering: feasibility and challenges. *ACM SIGMETRICS Performance Evaluation Review*, 38(3):56–60, 2011.

[40] Mohsen Jamali Langroodi, Joseph Peters, and Shervin Shirmohammadi. Complexity aware encoding of the motion compensation process of the h. 264/avc video coding standard. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, page 103. ACM, 2014.

[41] K. Lee, D. Chu, E. Cuervo, A. Wolman, and J. Flinn. Demo: Delorean: using speculation to enable low-latency continuous interaction for mobile cloud gaming. *in Proceedings of ACM MobiSys*, 2014.

[42] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement*, pages 1–14. ACM, 2010.

[43] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[44] Liang Liu, Hao Wang, Xue Liu, Xing Jin, Wen Bo He, Qing Bo Wang, and Ying Chen. Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 29–38. ACM, 2009.

[45] J.N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, pages 6–es. ACM, 2007.

[46] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE Computer Society, 2009.

[47] Onlive. http://www.onlive.com/.

[48] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of ec2 cloud computing services for scientific computing. *Cloud Computing*, pages 115–131, 2010.

[49] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K.G. Shin. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.

[50] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198. ACM, 2012.

[51] Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. *ACM Trans. Archit. Code Optim.*, 11(2):17:1—-17:25, July 2014.

[52] J. Liu R. Shea. Understanding the impact of denial of service attacks on virtual machines. In *Quality of Service (IWQoS), 2012 IEEE 20th International Workshop on*. IEEE, 2012.

[53] RackSpace. http://www.rackspace.com/.

[54] Carlos Reaño, Antonio J Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S Quintana-Orti. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012.

[55] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 61–72. ACM, 2012.

[56] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42:95–103, July 2008.

[57] R. Shea and J. Liu. Network interface virtualization: challenges and solutions. *Network, IEEE*, 26(5):28–34, 2012.

[58] R. Shea and J. Liu. Cloud gaming: Architecture and performance. *IEEE Network*, 27(4):16–21, 2013.

[59] Ryan Shea, Di Fu, and Jiangchuan Liu. Cloud gaming: Understanding the support from advanced virtualization and hardware. *IEEE Trans. Circuits Syst. Video Techn.*, 25(12):2026–2037, 2015.

[60] Ryan Shea and Jiangchuan Liu. On GPU Pass-Through Performance for Cloud Gaming: Experiments and Analysis. In *Proceedings of Annual Workshop on Network and Systems Support for Games*, NetGames '13, pages 6:1—-6:6, Piscataway, NJ, USA, December 2013. IEEE Press.

[61] Ryan Shea, Feng Wang, Haiyang Wang, and Jiangchuan Liu. A deep investigation into network performance in virtual machine based cloud environments. In *INFOCOM, 2014 Proceedings IEEE*, pages 1285–1293. IEEE, 2014.

[62] Ryan Shea, Haiyang Wang, and Jiangchuan Liu. Power consumption of virtual machines with network transactions: Measurement and improvements. In *INFOCOM, 2014 Proceedings IEEE*, pages 1051–1059. IEEE, 2014.

[63] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on*, 61(6):804–816, 2012.

[64] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *USENIX NSDI*, volume 11, 2011.

[65] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41:275–287, March 2007.

[66] Rahul Urgaonkar, Ulas C Kozat, Ken Igarashi, and Michael J Neely. Dynamic resource allocation and power management in virtualized data centers. In *Network Operations and Management Symposium (NOMS)*, pages 479–486. IEEE, 2010.

[67] Werner Vogels. Beyond server consolidation. *Queue*, 6:20–26, January 2008.

[68] G. Wang and T.S.E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[69] Haining Wang, Cheng Jin, and Kang G. Shin. Defense against spoofed ip traffic using hop-count filtering. *IEEE/ACM Trans. Netw.*, 15:40–53, February 2007.

[70] J. Whiteaker, F. Schneider, and R. Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.

[71] Chao-Tung Yang, Hsien-Yi Wang, and Yu-Tso Liu. Using pci pass-through for gpu virtualization with cuda. In *Network and Parallel Computing*, pages 445–452. Springer, 2012.

[72] K. Ye, X. Jiang, S. Chen, D. Huang, and B. Wang. Analyzing and modeling the performance in xen-based virtual cluster environment. In *2010 12th IEEE International Conference on High Performance Computing and Communications*, pages 273–280. IEEE, 2010.

[73] Miao Yu, Chao Zhang, Zhengwei Qi, Jianguo Yao, Yin Wang, and Haibing Guan. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 203–214. ACM, 2013.

[74] Miao Yu, Chao Zhang, Zhengwei Qi, Jianguo Yao, Yin Wang, and Haibing Guan. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 203–214, New York, NY, USA, 2013. ACM.

[75] Chao Zhang, Zhengwei Qi, Jianguo Yao, Miao Yu, and Haibing Guan. vgasa: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming. 2013.

[76] Chao Zhang, Jianguo Yao, Zhengwei Qi, Miao Yu, and Haibing Guan. vGASA: Adaptive Scheduling Algorithm of Virtualized GPU Resource in Cloud Gaming. *Parallel and Distributed Systems, IEEE Transactions on*, 25(11):3036–3045, November 2014.

[77] Zhou Zhao, Kai Hwang, and Jose Villeta. Game Cloud Design with Virtualized CPU/GPU Servers and Initial Performance Results. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 23–30, New York, NY, USA, 2012. ACM.