IMPACT OF VIRTUAL MACHINE OVERHEAD ON TCP PERFORMANCE

---

by

Vinod Kumar Rajasekaran

Submitted to

the graduate faculty

of the Department of Computer Science

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

Computer Science

June 17, 2004

Clemson University

Clemson, SC 29634-1906

Abstract

A Virtual Machine Monitor (VMM) is software that creates efficient, isolated programming environments called Virtual Machines (VM) that provide users with the appearance of direct access to a dedicated machine. Recently, there is renewed interest in VMMs for PC based workstations. VM technology has been proven useful in numerous scenarios including server consolidation, web hosting, software testing, and education. The grid and Web services communities are looking at VMs as a possible approach for dynamically managing computation services.

Although the performance of VMs has been widely studied, the effects of VMs on network performance have not been investigated. This thesis investigates the negative impacts of virtualization overhead on TCP performance. We show that the virtualization overhead can compress an ACK stream destined for a VM resulting in bursty send behavior and the effect becomes extreme as the number of VMs active on a host system increases. We also consider the impact that the virtualization overhead has on network performance using a realistic scenario involving a farm of VM Web servers. We show that under heavy loads, as the number of active VMs increase, network performance can deteriorate significantly as compared to a non-VM system subject to the same load.

Acknowledgments

I thank Dr Martin, my advisor for his guidance and support throughout my stay at Clemson University. It was a great opportunity to work with him on this project. I also thank Dr. Westall for his help at crucial stages of the project. His monitoring tool was extremely helpful in finishing my thesis on time.

A special thanks to Hima for her encouragement and support. Also, thanks to my wonderful roommates for the cheerful respite at the end of every day. Finally, I thank my parents for all their advice and words of encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

Chapter 1

Introduction

The concept of virtual machines was originally conceived by IBM in the 60's for their mainframe class of computers. It was an alternative approach to the conventional time-sharing operating systems. A virtual machine facility emulates multiple copies of the underlying physical hardware platform to higher layers. Each copy is fully isolated and secure from the others. In this way it allows multiple operating systems to simultaneously share a single hardware platform.

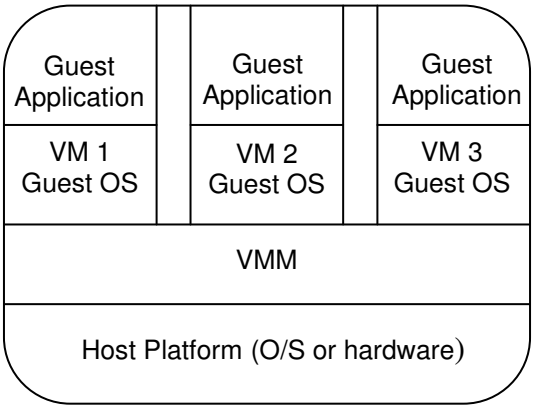

Figure 1.1 Virtual Machine System

The structure of a traditional virtual machine system is shown in Figure 1.1. The virtual machine monitor (VMM) is a software layer that virtualizes the underlying

hardware resources. It provides the necessary abstraction to efficiently multiplex the physical hardware across several instances of one or more operating systems. The hardware abstraction that the VMM exports is called a virtual machine (VM). A virtual machine provides an illusion of exclusive access of the computer hardware on a per user basis.

Lately, there has been renewed interest in the VM technology fueled by industrial trends including server consolidation, shared-memory multiprocessing and low cost PC based distributed computing. Virtual Machine solutions for PC platforms such as VMware [34], Virtual PC [33], and User Mode Linux (UML) [10] can provide significant cost savings to corporations. For instance, UML based Linux Virtual Servers (LVS) are used by redwoodvirtual.com and other service providers to host web servers and mail servers.

Since a VMM introduces additional layers of abstraction, performance of VMs has been a recurring concern. The layers of abstraction inherent in a VM can add up to an order of magnitude increase in overhead [15]. Further, there are several technical and pragmatic hurdles that arise when virtualizing the PC platform, which can impose serious performance obstacles [31]. The majority of related research has focused on the performance of VMM based systems. To the best of our knowledge, no work has been done in assessing the performance impact on a network by VMs. Because of the widespread use of VMs in the Internet, it is crucial to understand the impact that these systems can have on the Internet.

In this research, we study the impact of virtualization overhead on the TCP performance. Based on past studies on TCP dynamics [3, 17, 22, 23] and performance of VM systems [12, 19, 31], we conjectured that a VM environment, subject to heavy loads, could negatively impact TCP performance. Consider two systems, one a VM-based system configured to run multiple virtual Web servers and a second that does not have VMs. If both the systems are subject to identical loads (i.e., the same number of Web users), we show that the VM system can result in significantly worse end-to-end performance than the non-VM system. The objective of this thesis is to show this result and provide an intuitive explanation of this behavior.

Our evaluation is a two-step process. First, we show that the VM environment does affect the TCP self-clocking mechanism. We do this by looking at TCP dynamics at a host system based on VMware's GSX server [34]. We show that virtualization can induce ACK-compression in TCP connections involving a VM. Second, using a realistic scenario involving a farm of virtual Web-servers, we study network dynamics and the subsequent impact of TCP performance when a WAN is subject to varying traffic loads and numbers of active VMs. The contribution of our research is: 1) to show that the level of ACK-compression increases with the number of virtual machines running; 2) to show that the burstiness of TCP connections and the subsequent impact on end-to-end performance increase with the number of VM hosted Web-servers.

We chose VMware's GSX Server as our VMM system because of its wide-spread usage in the IT community. Also, we preferred a stable commercial VM product rather than an open source based system. Prior performance studies of VMware systems exist

[12, 19, 31], which allowed us to calibrate our methods. In the future we plan to extend our study by including User Mode Linux and Xen [5].

This thesis is organized as follows: Chapter 2 gives an overview of VM technology and VMware's GSX Server. The network testbed used for our analysis is described in Chapter 3. Chapter 4 presents the results of our analysis on the impact of VM overhead on a TCP ACK stream. Chapter 5 presents our results on our analysis on the network dynamics of a VM hosted farm of Web-servers. Chapter 6 identifies future directions of this analysis and suggests changes that could be made to alleviate the negative impacts of VM overhead.

Chapter 2

Overview

2.1 Virtual Machine Technology

An operating system is the foundation software that controls the functioning of the physical hardware resources. It creates a layer of abstraction on top of it, and shares the resources among higher-level software programs. A modern operating system uses the multiprogramming resource-sharing principle. A process is the fundamental unit of abstraction provided by such operating systems. IBM pioneered another resource sharing principle in which the fundamental unit of abstraction provided by the operating system is another "virtualized" machine. The foundation software that provides such an abstraction is called a Virtual Machine Monitor (VMM).

A VMM is defined as a piece of software for a computer system that creates efficient, isolated programming environments that provides users the appearance of direct access to a dedicated machine [27]. The programming environment created by the VMM is called a Virtual Machine (VM). VMs were originally developed by IBM for their main-frame class of computers such as the IBM 360 series and CMS virtual machines [15]. A virtual machine provides a fully protected, dedicated copy of the underlying physical machine's hardware [31]. Each virtual machine provides the illusion of exclusive access to the computer hardware.

2.2 VMM Requirements

A VMM system is fundamentally different from a modern time-sharing operating system. The properties required by a VMM system are described in [25, 27]. It includes three basic concepts: First, a VMM provides an execution environment which is "essentially identical" to the original machine. The environment is identical in the sense that any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the un-virtualized machine directly. However, there could be exceptions due to differences in system resource availability, and timing dependencies imposed by the VMM abstraction.

Secondly, the VMM must be in complete control of the system's resources allocated to the virtual machines. VMs should not be able to access resources that are not explicitly allocated to the VM by the VMM and further, a VMM should be able to regain control of the resources allocated to the VM. This characteristic ensures security and isolation across VMs. In essence, every VM is sand-boxed and the failure of one VM should not affect the functioning of other VMs.

Third, the VMM must be efficient. One approach to efficiency is for the majority of the virtual machine's instructions being executed on the underlying hardware directly, eliminating the overhead caused by virtualization. Some instructions, such as privileged instructions that cannot run directly can be emulated by the VMM.

2.3 Classification of VMMs

VMMs can be classified in many different ways [19]. This section describes two such classifications. The first one is based on how closely the emulated hardware interface matches the actual hardware. VMMs like VMware ESX Server [34], VMware GSX Server, and VMware Workstation [34] that are based on the x86 processors emulate a hardware interface for the virtual machines that is identical to the underlying hardware. On the other hand, Simulators like Bochs [6], Simics [26] and Xen provide a hardware abstraction that is different from the actual hardware.

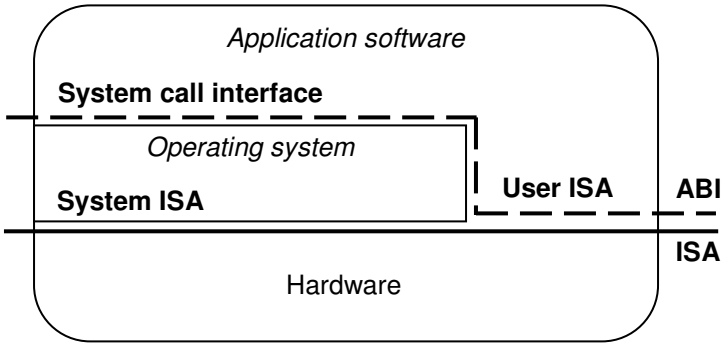

Figure 2.1: Important system interfaces

The second method of classification categorizes the virtual machines based on the "host platform" on which the VMM is implemented. Unlike the traditional VMMs that were pioneered by IBM, it is now possible to implement a VMM system using the interface provided by the operating system. In a computer system, there are two key

interfaces [30]: the Instruction Set Architecture (ISA) and the Application Binary Interface (ABI) (Figure 2.1). A combination of the two interfaces could be provided by a host platform for the creation and execution of virtual machines.

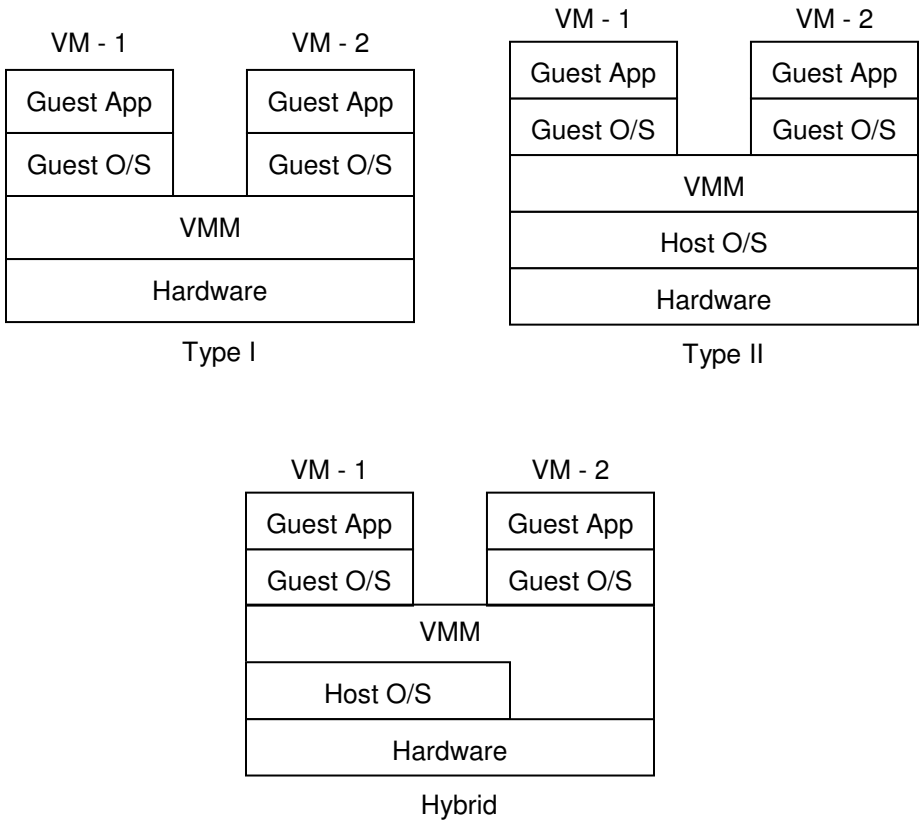

Figure 2.2 Block diagrams of VMM architectures

A second way to classify VMs is by Type I and Type II VMM. Type I VMMs run directly on the physical hardware or the ISA interface. A Type II VMM co-exists with a host OS [14]. They run on top of a host operating system as a normal application and

rely on the host O/S for hardware device support and resource allocation. A hybrid VM is an implementation that uses both the ISA and ABI interfaces for its existence. The block diagrams for type I, type II and hybrid VMs are shown in Figure 2.2.

IBM's VM/370, Disco [8], Xen and VMware ESX server are examples of Type I VMs. SimOS [29], User Mode Linux, FreeVSD [13], and Plex86 [24] are examples of Type II VMs. VMware GSX server and VMware Workstation are examples of hybrid VMs.

2.4 VM applications

The ability to run multiple operating systems simultaneously on the same hardware and to dynamically multiplex hardware between several virtual machines in a fair way make the virtual machines applicable to a wide range of scientific and commercial applications. The economics are driving the movement towards techniques such as server consolidation, shared-memory multiprocessing and low cost PC based distributed computing. Products like VMware and Virtual PC enable a corporation to consolidate under-utilized servers onto a smaller number of physical machines and thereby reduce support and management costs. Open source options like User Mode Linux and Xen make such cost-cutting technologies available even for very small corporations.

VMMs provide strong isolation across virtual machines sharing the same physical resource. Multiple layers of abstraction in a VMM system make it difficult for an intruder to breach system security. VMs are increasingly being used in the areas of content distribution networks [2], and intrusion analysis [11].

VMs are also being used in the areas of software development and testing, VMs provide a sand-boxed and controlled environment useful for fault injection [7] and Kernel/process debugging applications. In the academic environment, VMs can be a powerful educational tool by being incorporated in operating system, and IT courses. For example, VMMs can be used to create virtual networks providing a versatile testbed environment for intermediate to advanced network courses [21].

Future IT requirements are likely to include on-demand resource provisioning. Virtual machines have the potential to play a vital role as they can be dynamically created, configured and moved around networked systems like grids and clusters. Virtual machines can provide dynamic multiplexing of users onto physical resources with a granularity of single user per operating system session [5, 9, 18, 36].

2.5 Performance of Virtual machines

Performance is a recurring concern in every application of virtual machine technology. The layers of abstraction inherent in a VM can add significant overhead, possibly up to an order of magnitude [15]. In this section, we first describe the functional modules of a VMM system to help us understand the working of a VMM. Then we point out the difficulties in implementing a VMM for PC hardware as background for understanding the performance penalties inherent in such an implementation. We describe only the PC platform, because we used VMware GSX Server, an Intel's x86 based VMM system for our research. Next, we summarize the related work in the performance of virtual machines and outline the significance of our research.

2.5.1 VMM modules

In general, a VMM has three kinds of functional modules. They are referred as the dispatcher, allocator, and interpreter modules [25]. The dispatcher modules are the top-level control programs. All hardware traps that occur in the context of a virtual machine are made to a dispatcher code. The dispatcher invokes the correct module to process the event that caused the trap.

The allocator modules are responsible for resource allocation and accounting. An allocator ensures that there are no resource conflicts in terms of security across VMs. Whenever a VM tries to execute a privileged instruction that would have the effect of changing the machine resources associated with that VM, a trap is made to the dispatcher. The dispatcher in turn calls the allocator to allocate the requested resource and modify the resource in a controlled manner.

The final module type is the interpreter. Depending on the classification, a VMM will not be able to directly execute a set of instructions on the real processor. Interpreter modules are routines that simulate the execution of such instructions at the software level.

2.5.2 Difficulties in virtualizing the PC platform

There are several technical hurdles in virtualizing an Intel's x86 PC platform [27, 31]. These hurdles, in turn, have severe implications on the overall performance of an x86 based virtual machine system. Several well known problems are described below.

When executing an application inside a virtual machine, some processor instructions cannot be directly executed on the underlying physical processor. These

instructions would interfere with the state of the VMM or the host O/S and hence might affect the stability and integrity of the whole system. Such instructions are called *sensitive* instructions. A processor is *virtualizable* only if the set of sensitive instructions for that processor is a subset of the set of privileged instructions [25]. The Intel's x86 processors support sensitive instructions that can be run on the unprivileged mode (the user mode) and hence the architecture is not naturally virtualizable [27, 31].

A VMM is usually run in the privileged mode and the VMs are run in user mode. When a VM executes a sensitive instruction that is privileged, a trap occurs and the dispatcher module of the VMM takes over. Correct execution of the unprivileged-sensitive instructions involves identifying such instructions in an instruction stream and software level emulation of the same. The whole process is expensive in term of time and hence is a potential source of performance bottlenecks.

Secondly, there is a large diversity of devices that are found in PCs. This is due to the PC's open architecture. It is difficult (as well expensive) to provide device drivers in the VMM for all supported PC devices. A common solution to this problem would be to use the Type II or hybrid VMs, wherein a host operating system supplies the necessary device drivers the VM guest must use. Obviously, this approach involves additional CPU overhead in comparison with Type I VMMs.

2.5.3 Related work

In [31] the authors have done an analysis on the performance of the virtual NIC on a VMware workstation based VMM system. In their analysis, they have described the working of the virtual network subsystem in great detail. The authors have pointed out

that the high frequency of *world switches* (section 2.6.3) is the major source of overhead when a VM accesses a network interface card. They have suggested optimizations like send-combining (section 2.6.3), IRQ notification and Handling I/O ports in the VMM that would optimize the CPU utilization for such devices.

In [19], the authors have shown that Type II VMMs are an order of magnitude slower than running directly on the host machine (no vm), and much slower than Type I VMMs. The authors suggest a set of optimizations that reduces the virtualization overheads to 14-35% relative to running on a standalone host (no VM running). The authors also performed a set of performance tests with VMware Workstation 3.1 and found that the system could saturate a 100 Mbps network on a moderately powered PC.

In [12], the authors have performed a series of macro and micro-benchmark tests on a VMware Workstation 3.0a based system. They report that the execution time overhead is less than 10% for CPU-intensive tasks that include network transfers for the benchmark suites.

All of the previous work about VMM performance analyzes the internals of a VMM system. In this thesis we have looked at the performance of TCP when virtual machines are used to host servers. We study the impact that VMM virtualization can have on a TCP connection's performance. To the best of our knowledge, little or no work has been done in assessing the performance VMM based system with this perspective.

2.6 VMware GSX Server

We used VMware GSX server 2.5.1. The VMware GSX Server has been designed for enterprises that want to consolidate multiple virtual servers on one high-end Intel-based system. It supports as many as 64 VMs running on hosts that have up to 32 processors and 64GB of RAM. VMware provides sophisticated networking support to virtual machines. The following subsections outline the underlying architecture and the networking capabilities of VMware GSX server. Since it is proprietary software, we do not have the details on the actual implementation of the networking module. However, in reference to [31] the details of the working of VMware Workstation product's networking module are outlined in section 2.6.3. The VMware GSX server and the Workstation version are based on the same VMM architecture and hence the details provided here give an understanding of the networking subsystem.

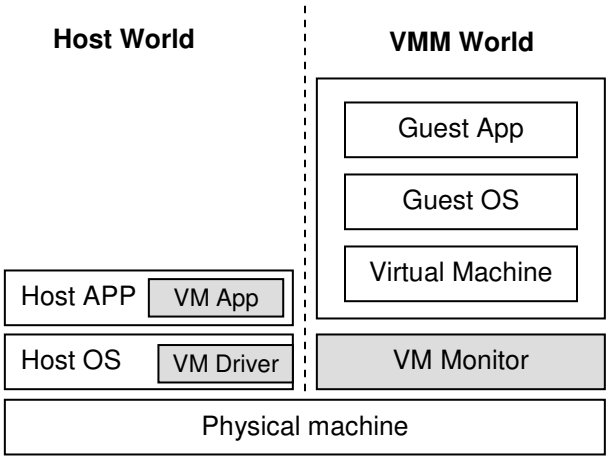

Figure 2.3 VMware's Hosted VMM architecture

2.6.1 Hosted Virtual Machine Architecture

VMware GSX server is based on VMware's hosted machine architecture (Hybrid VMM). The hosted virtual-machine architecture allows the VMM to co-exist with a pre-existing host operating system, and rely upon that operating system for device support. The architecture is shown in Figure 2.3.

VMware's hosted architecture has three major components: the VMApp, VMDriver and the VMM (Figure 2.3). A virtual machine can be started in user mode. When it is started, the application component, VMApp, uses the VMDriver that is loaded into the host operating system during boot-up to establish the privileged VMM component. The VMM component runs directly on hardware. Once the VMM component is established, the physical processor is executing either in the host world or the VMM world [31]. The VMDriver facilitates a context switch between the two worlds.

The operating system running on the virtual machine is referred as the guest operating system and the applications that they run are referred to as guest applications. When a guest application performs pure computation, it runs like any other user-level process. The VMM executes those virtual instructions on the hardware directly without making a switch to the host operating system. When a guest performs an I/O operation like reading a file, or transmitting a packet across the physical network, the VMM initiates a world switch. A world switch involves saving and restoring all user and system visible state on the CPU, and is thus more heavyweight than a normal process context switch [31]. Following the switch to the host world, the VMApp performs the requested I/O operation on behalf of the VM through appropriate system calls to the host machine.

For example, reading a disk block for a VM is done by issuing a read() system call in the host world. Similarly, if a hardware interrupt occurs while executing in the VMM world, a world switch occurs and the interrupt is reasserted in the host world so that the host OS can process the interrupt as if it came directly from the hardware.

This architecture deals with the vast array of PC hardware by relying on the host operating system. However, it is obvious that there would be some performance degradation associated with this architecture. For example, an I/O intensive application would become CPU bound due to excessive world switches.

2.6.2 Networking Capabilities

VMware GSX server provides several networking components that can be used to build a wide range of network topologies. It also provides three preconfigured network configurations that are commonly used. This subsection provides an overview of the various networking components and options that comes with the GSX server [34].

The networking components include virtual switches, virtual network adapters, and a virtual DHCP server. The core component virtual switch, emulates a physical switch. Multiple machines can be connected to a virtual switch. The GSX server supports up to 9 virtual switches on a Windows host and up to 99 virtual switches on a Linux host. The switches are named as vmnet0, vmnet1, and so on. Three of those switches are reserved for the preconfigured network setups: the vmnet0, vmnet1 and vmnet8. These are called *Bridged, Host-only* and *NAT* respectively.

Virtual network adapters appear in the same way as a physical Ethernet adapter appears on a machine. The virtual device is an AMD PCNET PCI adapter. Each virtual

machine can have up to 3 virtual network adapters. These virtual adapters can be connected to any of the virtual switches.

The virtual DHCP server is created automatically for two of the pre-configured network setups namely, the host-only and NAT configurations. It provides IP network addresses to the virtual machines that are connected to vmnet1 and vment8 switches. The pre-configured network-setups are described below:

Bridged Networking:

This configuration connects the virtual machine to the LAN used by the host. The virtual machine appears on the LAN as if it is another physical machine present on the LAN. The vmnet0 device on the host maps the virtual network adapter to the host machine's network adapter and acts like a true bridge. Each virtual machine must be assigned a unique IP address on the network. Bridged networking can be setup with any of the host machine's wired Ethernet adapters.

NAT Networking:

This configuration creates a "virtual" private network of VMs that use NAT (Network Address Translation) to communicate with the outside world. To do this setup, the VM's virtual NIC should be connected to the virtual switch named vmnet8. Using NAT, the machines in the virtual private network can access the machines on the network to which the host machine is connected. GSX server automatically runs a virtual DHCP server for the private network and also configures the IP network number for the private network to an unused private IP network number. The GSX Server also gives an option of manually

configuring the virtual DHCP server with the desired IP network number. Further, the vmnet8 switch appears as a separate network interface card in the host machine. The IP address of this interface would the host '.1' of the private network. It is an abstraction of the host machine to the virtual machines that are part of the private network. In other words, the virtual machines on the private network see the host machine as another virtual machine present in the private network.

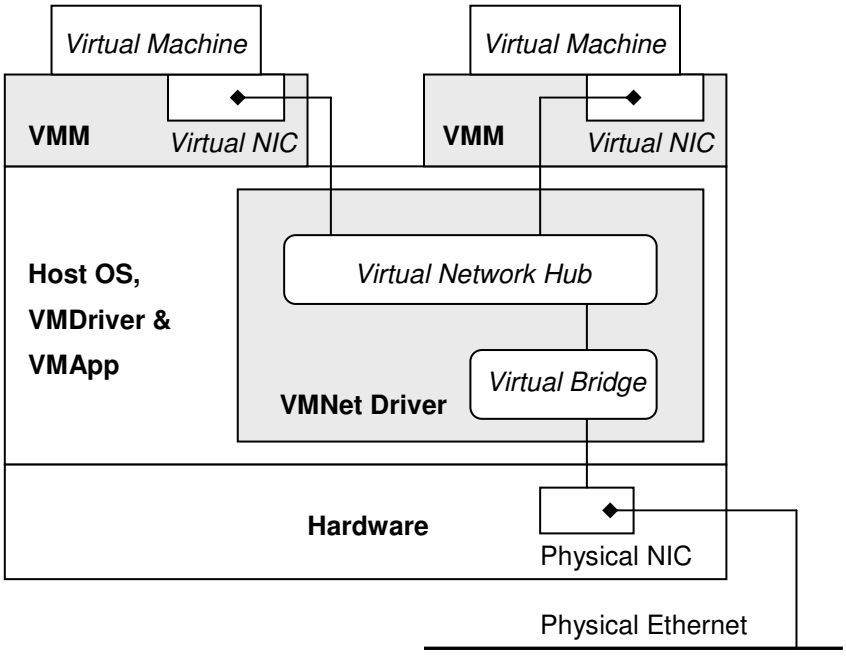

Figure 2.4 Virtual Bridge Components

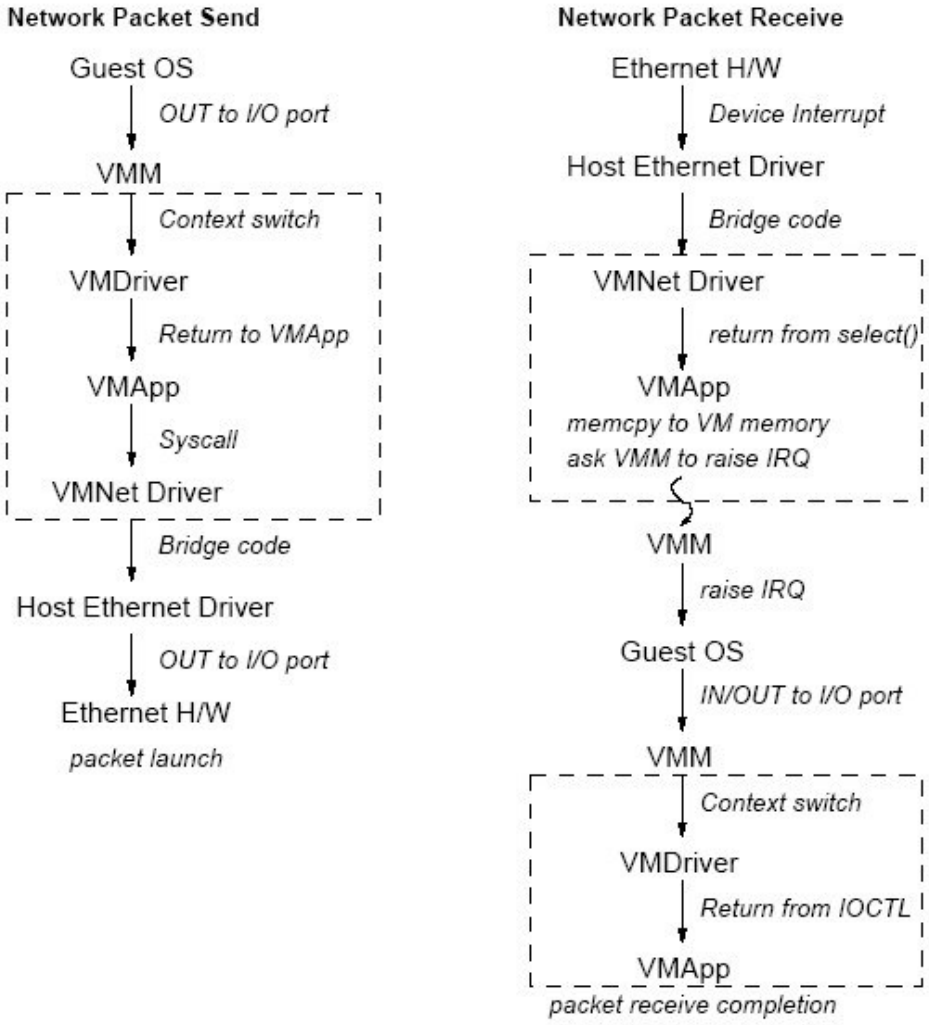

Figure 2.5 Components involved in a VM network packet send and receive

Host-only Networking:

The host-only networking configuration is also used to create a virtual private network comprising of the host machine and the virtual machines. The setup is similar to the NAT configuration except that the VMs will not be able to access the host machine's network.

To establish a connection between the virtual private network and the outside world, routing or proxy software has to be set-up on the host machine.

2.6.3 Functioning of a Bridged Network Setup

The components of a virtual bridged system are shown in the Figure 2.4. The virtual NICs are bridged to the physical NIC through the virtual bridge that is configured during the installation of VMware. Each virtual NIC has its own MAC address and appears like a physical NIC to other systems on the same network. The physical NIC is run in promiscuous mode to capture all the packets on the network. The VMNet driver performs a proxy ARP function on behalf of the virtual NICs. Within the VM, the virtual NIC appears as a fully-functional AMD PCNET PCI network interface card.

Figure 2.5, extracted from [31], shows the components involved when sending and receiving packets through the virtual NIC. The boxed components represent the additional work due to the virtualization. We can see that, a packet send involves at least one world switch and a packet receive process involves at least two work switches. This extra overhead consumes large number of CPU cycles and in effect increases the load on the CPU. For high throughput connections, it is likely that the system will be CPU bound.

2.7 ACK-compression

In equilibrium, the TCP sender maintains a sliding window of outgoing full size segments. Every ACK arriving at the sender causes the sender to advance the window by the amount of data just acknowledged. The pre-buffered data segments are sent as soon as the sending window opens up. Ideally ACKs arrive at the TCP sender at the same rate

at which the data packets leave the sender. This leads to the self-clocking nature of TCP, where the sending rate never exceeds the bottleneck link speed [17]. However, the clocking mechanism will be disturbed by delay variations in either the forward or backward directions. The delay variation can lead to compression or decompression of the ACK or segment stream. One important side effect of this is that an estimate of the bottleneck link speed by either the sender or receiver is likely to be incorrect. Timing compression in particular is detrimental as it can cause packet loss.

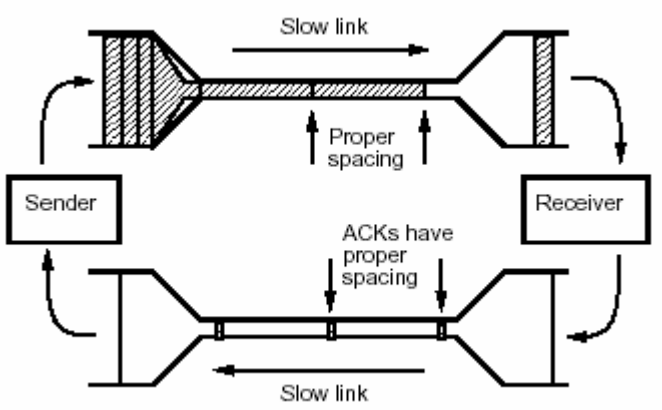

Figure 2.6: Steady State TCP Behavior

Paxson [23] has shown that three types of timing compression are possible with a TCP session: *ACK-compression, data packet compression,* and *receiver compression.* ACK-compression occurs when the spacing of successive acknowledgements is compressed in time while they are in transit. Data packet compression occurs when a

flight of data is sent at a rate higher than the bottleneck link rate due to sudden advance in the receiver's offered window. Receiver compression occurs when the receiver delays in generating ACKs in response to incoming data packets, and then generates a whole series of ACKs at one time.

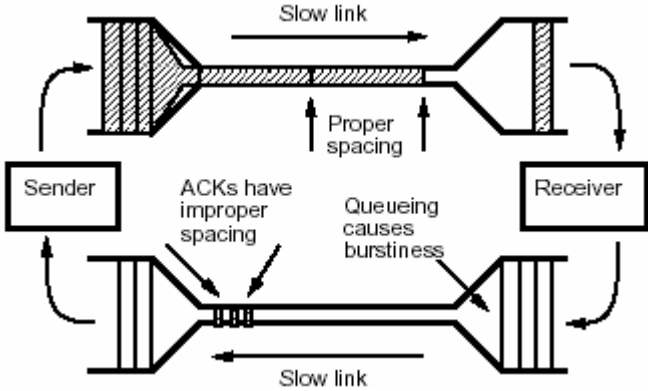

Figure 2.7: ACK-compression

Of the three, ACK-compression is the most common form of compression in a real network. It has been shown that ACK-compression does occur in the Internet typically due to congestion in the reverse path [16, 23]. An end-to-end path traversing multiple nodes can be modeled by representing the sending node and receiving node connected by the bottleneck (slowest) link. Figures 2.6, 2.7 extracted from [22] show two ends of a TCP session and the bottleneck link. Figure 2.6 shows the ideal case where the spacing between successive ACKs is preserved allowing the TCP sender to match rather

than overrun the bottleneck link capacity. Figure 2.7 demonstrates ACK-compression. The ACKs arrive at the TCP sender compressed as the spacing between successive ACKs is now smaller than it was when originally sent by the sender. In [23], the author has shown that if ACK-compression is frequent, it presents two problems. First, as ACKs arrive they advance TCP's sliding window and clock out new data packets at the rate reflected by their arrival. As a result TCP send dynamics becomes bursty. The data packets go out faster as a burst which can lead to poor network performance. Second, sender-based bottleneck estimation techniques can misinterpret compressed ACKs as reflecting greater bandwidth than truly available. This could lead to congestion and loss of efficiency.

Chapter 3

Experimental Setup

3.1 Network testbed

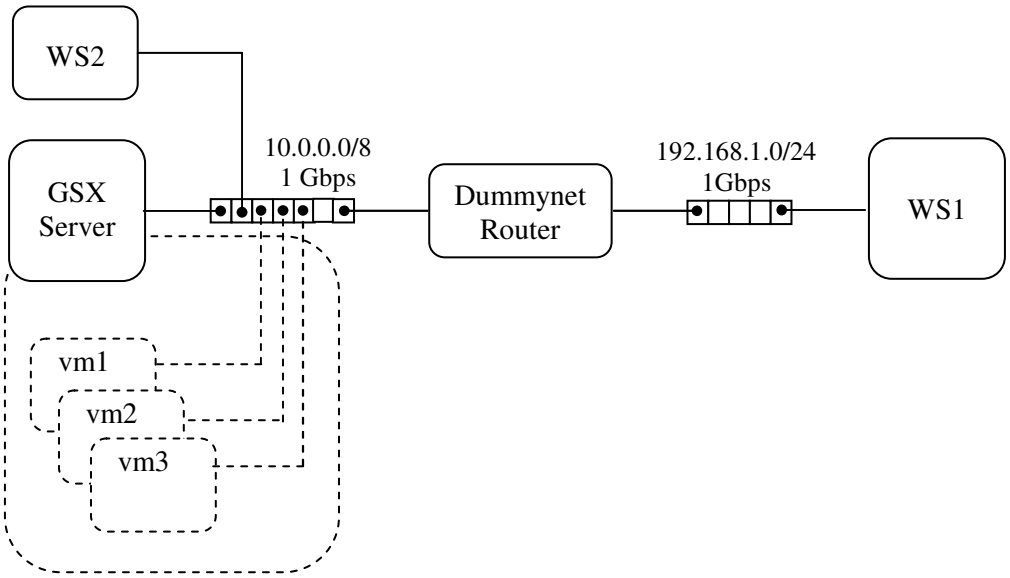

Figure 3.1 Network testbed

The network testbed that we used for our experimentation is shown in Figure 3.1. The testbed is made up of two separate local area networks (10.0.0.0/8 and 192.168.1.0/24) interconnected by a router. The configuration models a Web server farm that might have a combination of physical and virtual Web servers each sharing a single WAN connection for Internet connectivity. The machine labeled GSX server hosts our

installation of the VMware GSX server. It is a Dell Precision 450n workstation equipped with 2 GBytes of RAM and dual Intel Xenon processors running at 2.4GHz with hyper-threading. As an example, Figure 3.1 shows a configuration including three virtual machines.

The router is a Dell Optilex GX250 PC equipped with 512 MBytes of RAM and an Intel Pentium IV processor running at 2.4GHz. It runs FreeBSD 4.8 and uses *dummynet* [26] to control path latency and bandwidth in order to emulate a WAN environment. In [26], the author shows that *dummynet* introduces minimal overhead and can be used for accurate simulation of real-world networks with bandwidth and delay limitations.

The machine labeled WS1 is identical to the VMware GSX server except that it is equipped with only 1 GB of RAM. WS1 runs RedHat 8.0 Linux distribution with smp kernel 2.4.20. The machine labeled WS2 is physically identical to the router. WS2 runs RedHat 8.0 Linux distribution with kernel 2.4.20. These machines are used as the traffic generators for the virtual machines during the experiments.

3.2 VMware GSX server setup

The VMware GSX Server version 2.5.1 [34] is used to create the virtual machine environment for our study. RedHat 8.0 Linux distribution with smp kernel 2.4.20 is our host operating system. The *VMware tools* that comes with the VMware GSX server was also installed.

Seven virtual machines were created using the VMware GSX server. They are configured with 256 Mbytes of RAM, 4 GBytes virtual hard disk, and one virtual Ethernet network adapter. The GSX server is configured to run the bridged networking setup for all the virtual machines so that, each VM has a unique 10.0.0.0/8 IP address and appear on the 10.0.0.0/8 network as a unique host. The guest operating system is RedHat 8.0 Linux distribution with kernel a 2.4.20.

The Apache [1] server version 2.0 is run on each of the virtual machines. An instance of Apache is also installed on the machine running VMware GSX server to do the baseline run.

3.3 Traffic sources

We use Scalable URL Reference Generator (SURGE) [4], which is a software that is used to simulate the behavior of a large number of Web clients. It is a realistic http workload generator. Several processes and threads are spawned to simulate the behavior of multiple Web clients. The execution of each thread mimics the traffic pattern generated by an individual Web user. The sequence of URL requests that are generated exhibit representative distributions for document popularity, document sizes, request sizes, temporal locality, spatial locality, embedded document count and off times.

The SURGE architecture is split into three parts: the client set up, the server set up and the client request generator. The client set up portion generates the sequence of URL requests which would match a realistic workload distribution. The setup parameters are that were used for our experiments is shown in Table 3.1.

A file called *mout.txt* is generated during the client setup process which contains a mapping of the client requests and the size of the requests. It is then used to setup the server with Web objects. The client request generator is the executable that is run in order to mimic the realistic Web client behavior. The parameters to the request generator control the amount of workload being generated.

Table 3.1: SURGE setup parameters

HTTP version	1.0
Zipf	20000, 2000
Sizes	2000
match	1

We also use a simple TCP data transfer program to benchmark network performance. The application establishes a TCP connection between two IP addresses. The client sends data packets to the server of the size specified by the user. The server consumes all the data packets sent by the client. The data transferred is one of more copies of an in-memory buffer. Because no disk access is required at either endpoint, the application is capable of sending data at high speeds.

Chapter 4

VMware side Analysis

In this chapter, we show that the ACK stream destined for a virtual machine is compressed in time due to the virtualization overhead and further, that the compression is more pronounced with the increase in the number of active virtual machines.

4.1 Network Configuration

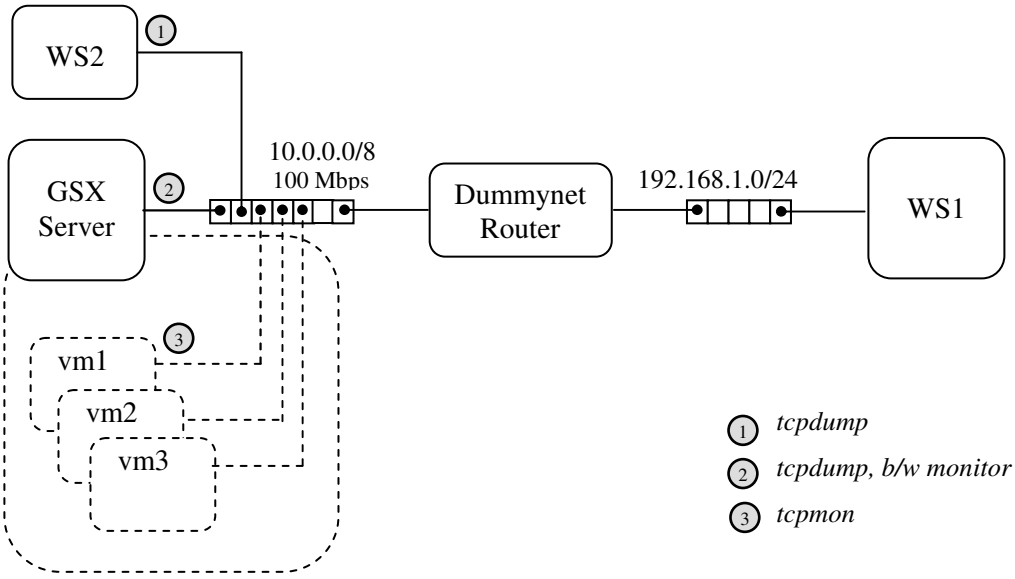

Figure 4.1 Network Configuration for VMware side analysis

The network configuration used for this analysis is shown in figure 4.1. Only the 10.0.0.0/8 LAN is used for this set of experiments. A 100 Mbits/sec switch is used in the 10.0.0.0/8 LAN. All the machines in the 10.0.0.0/8 network are equipped with Gigabit network interface cards but will dynamically downgrade to Fast Ethernet operation. We used a 100 Mbits/sec network setup rather than a gigabit network for the following reasons: 1) to run the test connection for an extended period, 2) to avoid excessive interrupt rate at the host, and 3) to reduce the overhead of our timing mechanism at the test virtual machine.

4.2 Experimental Methodology

In this set of experiments, we study the impact of virtualization overhead on the TCP acknowledgement stream. First, we study the impact of virtualization on the ACK stream in a relatively simple setup. Only one virtual machine is started and the ACK stream is compared against the baseline experiment which does not use VMs. Secondly, the impact is studied as a function of the number of virtual machines running concurrently to observe and quantify the correlation between the number of virtual machines and the subsequent impact. The virtual machine labeled vm1 is the test machine for this study. A single TCP connection is established between vm1 and WS2. An application on vm1 sends about 150 Mbytes resulting in roughly 50,000 ACKs arriving at the GSX Server host.

The router is used as the source of background traffic for the other virtual machines. For the studies reported here, the background traffic was limited to a single

TCP connection per virtual machine that was similar to the test connection. The purpose of the background traffic was to increase the level of contention for the network interface card at the host machine. As the number of active virtual machines increase, the contention for the network card also increases though the aggregate network load is the same.

The ACK arrival distribution at the virtual machine is compared against the ACK arrival distribution at the GSX server, namely, the host machine, to quantify the impact of virtualization overhead on the ACK distribution. The study is conducted by varying the number of concurrently running virtual machines by subjecting the GSX server to the same amount of network load. The number of virtual machines used is {0, 1, 2, 3, 4, 5, 6, and 7}. The '0-vm' case serves as the baseline run, wherein no virtual machine is started. The unmodified host operating system on the GSX server is used in this case. The network load is measured as the aggregate outbound bandwidth at the GSX Server. The test TCP connections lasted for about 30 to 50 seconds of elapsed real time.

4.3 Performance data

Performance data is collected at multiple locations within the 10.0.0.0/8 network. The various vantage points of the tools are shown in Figure 4.1. For the data collected at the virtual machine, a customized kernel module called *tcpmon* [35] is used. A character device named *tcpm* is created in the test virtual machine. The tcpmon module is loaded as a Linux device driver to tcpm. The driver can capture data relating to opening and closing of TCP sessions or all the data packets pertaining to a TCP port. The option is configured

using the *ioctl()* system call interface provided by the driver. We configured the driver to capture all the data packets of our test connection between vm1 and WS2 based on the TCP port we used for the test connection. The data captured include the timestamp at which the packet was seen in the TCP/IP stack, the sequence number and the ack number in the packet, and the direction (outgoing/incoming) of the packet. The timestamps are obtained by reading the Pentium Processor's *Time Stamp Counter* (TSC) register [16] whenever the packet is seen by the port filter in the tcpmon program. The data is post-processed to obtain the ACK inter-arrival times.

The correctness of our analysis rests on the accuracy of the timing data reported by the *tcpmon* program. Initially, we used *tcpdump* [32] to collect data at the VM. But, we found that the timing reported by *tcpdump* was incorrect. For instance, the elapsed time for a TCP session as reported by *tcpdump* at the VM was lesser than the actual elapsed time. We performed timing experiments with the RDTSC instruction and found that the timing was accurate. The elapsed time for a TCP connection at the host and the VM were comparable. Also, the overhead in reading the timing information from the TSC register is very minimal in the order of microseconds. The authors of [31] have used a similar methodology to obtain the timing information in their work with VMware workstation.

The ACK stream arriving at the host machine is captured by the *tcpdump* program in the Linux host operating system on the GSX server. The network load on the host is obtained by periodically accessing the */proc/net/dev* file of the GSX server and reading the packet counter for the server's network interface card.

4.4 Results and Analysis

In this section we show empirical evidence that the ACK stream destined for a virtual machine is compressed in time due to the virtualization overhead. Interestingly, we also show that the ACK stream is smoothed out. We also show that the level of ACK-compression increases with the increase in the number of active VMs.

First, we show that the aggregate network load that the GSX server was subject to during the various iterations of this set of experiments is the same. The upstream throughput seen at the GSX server is shown as a function of the number of virtual machines running concurrently in Figure 4.2.

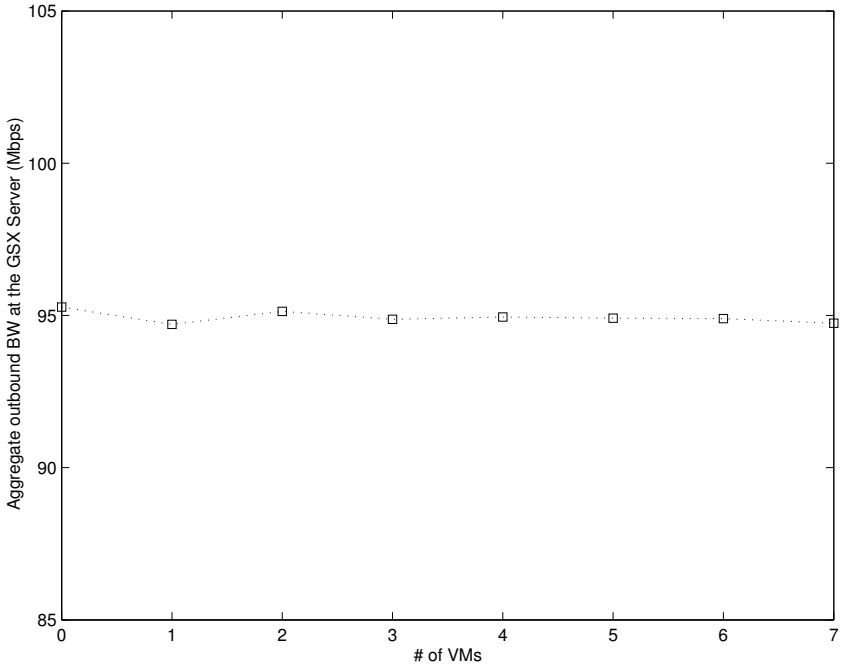

Figure 4.2: Test workload

The data shown is the actual departure rate at the GSX Server. Roughly 95 Mbps on sustained bandwidth was consumed which effectively saturated the 100 Mbps network. We can see that the network load remained the same irrespective of the number of virtual machines running concurrently. It was almost the same in the baseline experiment.

In our setup, the two hosts are on the same network interconnected by a 100 Mbps switch. Hence, there is no potential source of data or ACK-compression within the network. However, WS2 can induce receiver compression by delaying generating ACKs in response to incoming data packets, possibly because of system delays which leads to bursts of ACKs. Our focus however is on the ACK-compression that occurs at the VM system, which can induce much higher levels of ACK-compression.

Figure 4.3 shows the ACK inter-arrival distribution as seen at the host machine for our baseline test which did not use a virtual machine. From the figure we see that the inter-arrival times of the ACKs are aligned with the multiples of one-way packet transmit time over a 100 Mbps link (0.12ms). In our setup, Linux was “not” configured for TCP_NODELAY. So, normally two segments are acknowledged per ACK. Hence, a majority of the ACKs (37%) arrive at the GSX Server at double the packet transmission time over the 100Mbps link. However, about 20% of the ACKs arrive with a spacing of a one-way packet transmission time (0.12ms). This is due to two reasons: 1) q*uick-acknowledgements* in Linux, wherein an ACK is sent for every TCP segment [28] and 2) possible timing variations in the system. Roughly 8.9% of the ACKs arrive with a spacing less than 0.12ms inter-arrival time. This suggests ACK-compression. We

conjecture that this compression is caused by the receiver (WS2). We confirm this by comparing the distribution of ACK arrivals observed at the GSX server with that observed at WS2. We see the same frequency of ACKs that are sent with departure time less than 0.12ms. The other modes represent higher numbers of data segments being acknowledged per ACK. These are mostly caused by ACKs being dropped and/or system delays at WS2.

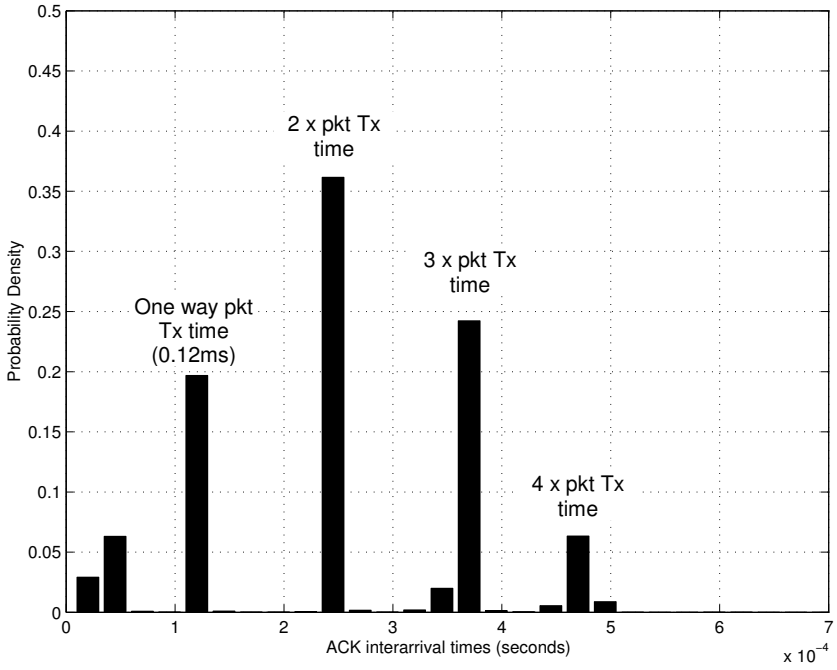

Figure 4.3: ACK inter-arrival distribution for the baseline test (no VM)

We are interested only in finding the extent that the ACKs are compressed due to the virtualization overhead. For this, we compare the ACK arrival distribution seen at the

host's interface and the ACK arrival distribution as seen on the virtual machine. In the process, first, we show that the virtualization overhead has considerable impact on the ACK stream by comparing our baseline run and the case with only one virtual machine running. Next, we show that a correlation exists between the number of virtual machines running concurrently and the extent of this virtualization impact on the ACK stream.

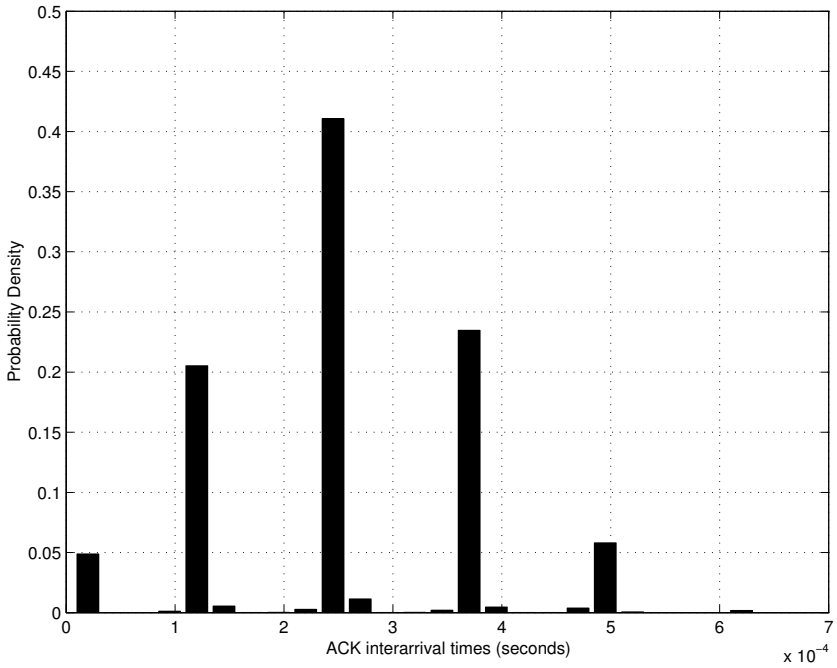

Figure 4.4: ACK inter-arrival distribution at host for 1 active VM

Figures 4.4 and 4.5 show the ACK inter-arrival distribution as seen at the host machine and at the VM respectively when only one virtual machine is started. Ideally, in the absence of VM effects, we would expect the ACK inter-arrival distribution seen at

vm1 for the 1vm case to be similar to the distribution observed at the host (Figure 4.4). However, Figures 4.4 and 4.5 show that the ACK arrival distribution is greatly perturbed by the virtualization overhead.

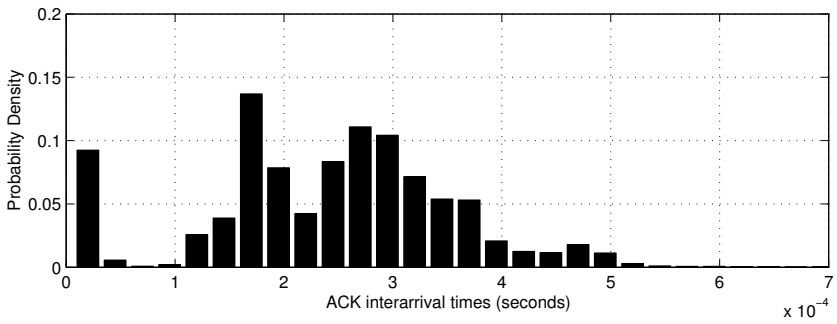

Figure 4.5: ACK inter-arrival distribution at vm1 for 1 active VM

The distribution shown in Figure 4.4 is very similar to that of the baseline (figure 4.3). This suggests the absence of other sources of compression in the network. Although, the network loads were the same for the baseline and when there was one VM active, the CPU loads were not the same. The average CPU idle time as reported by the *top* utility for the baseline was 92.25% and for 1 active VM it was 73.45%. As we will show later this additional CPU overhead caused by the VMs perturbs the ACK stream when the system is subject to a comparable network load. Figure 4.5 suggests that the ACK stream is affected in two different ways: 1) the ACK stream is compressed, and 2) the ACK inter-arrival distribution is also smoothed out.

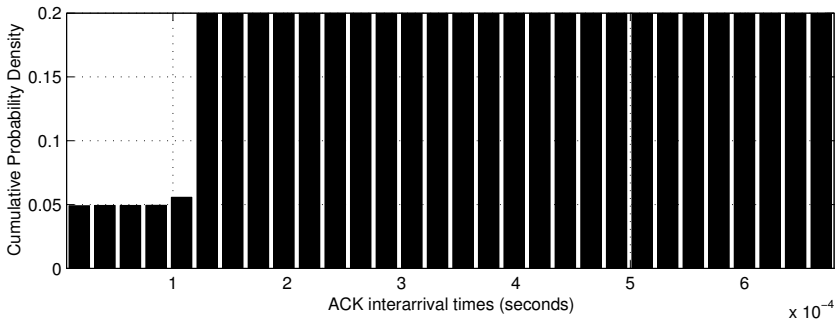

Figure 4.6: CDF of ACK inter-arrival distribution at the host for 1 active VM

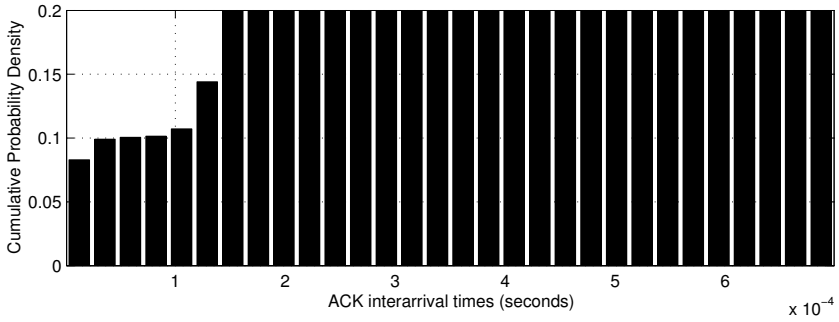

Figure 4.7: CDF of ACK inter-arrival distribution at vm1 for 1 active VM

Figures 4.6, 4.7 show the CDF of the ACK inter-arrival times of the sub 0.12ms window of the 1 vm case. It helps us to identify the extent to which the ACKs are compressed. Only 5% of the ACKs are compressed by the time they reach the host machine. However when the same ACK stream reaches the VM, about 11% of the ACKs have been compressed. ACK-compression can also be observed by looking at the other

end of the distribution. The modes at 0.36ms and 0.48ms shown in Figure 4.4 have been greatly reduced when the same ACK stream is observed at the VM Figure 4.5.

We attribute two possible reasons why ACKs are compressed on a VMM based system. First, the delivery of the ACK does not occur until the target virtual machine can be dispatched. Hence under heavy loads with multiple competing VMs the dispatch can be delayed greatly. The ACKs might get queued up at the host and be delivered in a burst when the target virtual machine is eventually dispatched. Hence we would expect an increased level of ACK-compression with the increase in the number of virtual machines. The results shown later in this section corroborates this conjecture. Secondly, in order to minimize the number of world switches and to improve the sustained throughput, VMM implementations delay the transmission of outbound packets (*send-combining* in VMware [31]). It is possible for an equivalent procedure to be applied to inbound packets resulting in ACKs being compressed in time. So an ACK arrives at the host and is deliberately delayed for a small amount of time in hopes that additional ACKs arrive within a very short time. We refer this as receive data combining in the remainder of this thesis. If so, all ACKs are sent to the VM requiring a single world switch.

Interestingly we also see that the ACK stream is smoothed out. The distribution seen at the host (Figure 4.4) shows that the arrivals are aligned along the multiples of one way packet transmission time. But, Figure 4.5 shows that the inter-arrival distribution at the VM is continuously spread over the timescale. The receipt of a data packet requires at least two world switches (section 2.6.3). As a result, there is additional processing delay due to virtualization. The delay is evident from the movement of the mode from 0.24ms

in Figure 4.4 to 0.26ns and 0.28ms in Figure 4.5. This delay in addition to the heuristics applied for optimization causes the ACK stream to be smoothed out.

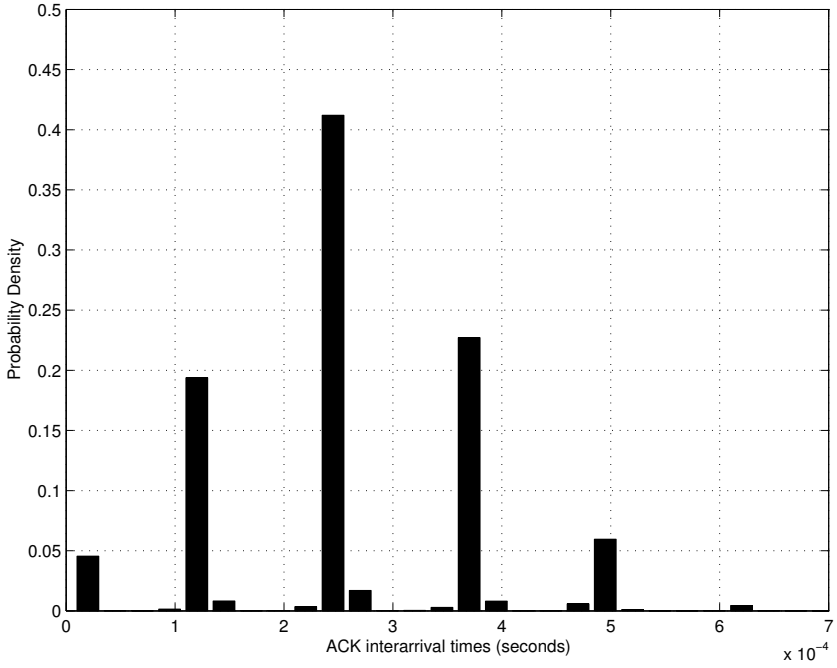

Figure 4.8: ACK inter-arrival distribution at host for 2 active VMs

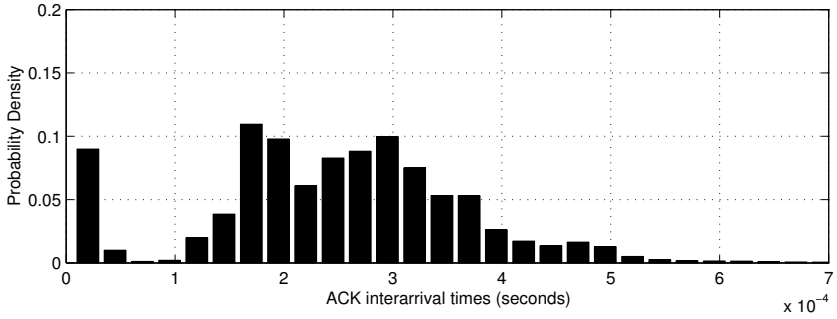

Figure 4.9: ACK inter-arrival distribution at vm1 for 2 active VMs

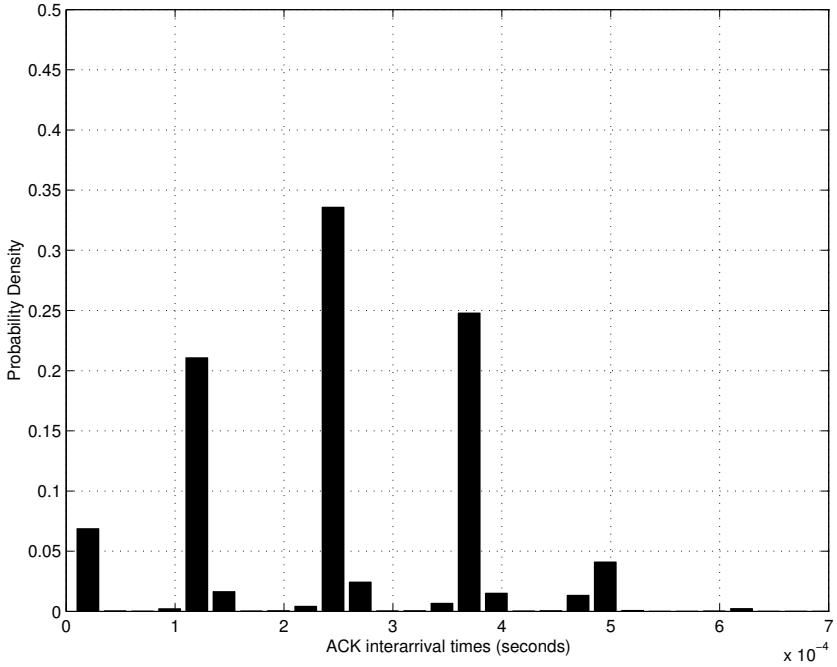

Figure 4.10: ACK inter-arrival distribution at host for 3 active VMs

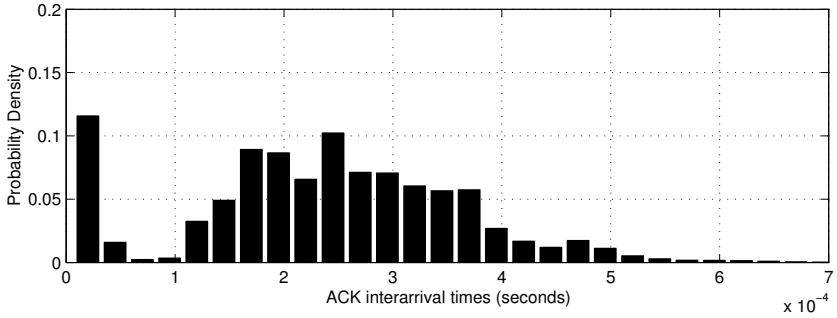

Figure 4.11: ACK inter-arrival distribution at vm1 for 3 active VMs

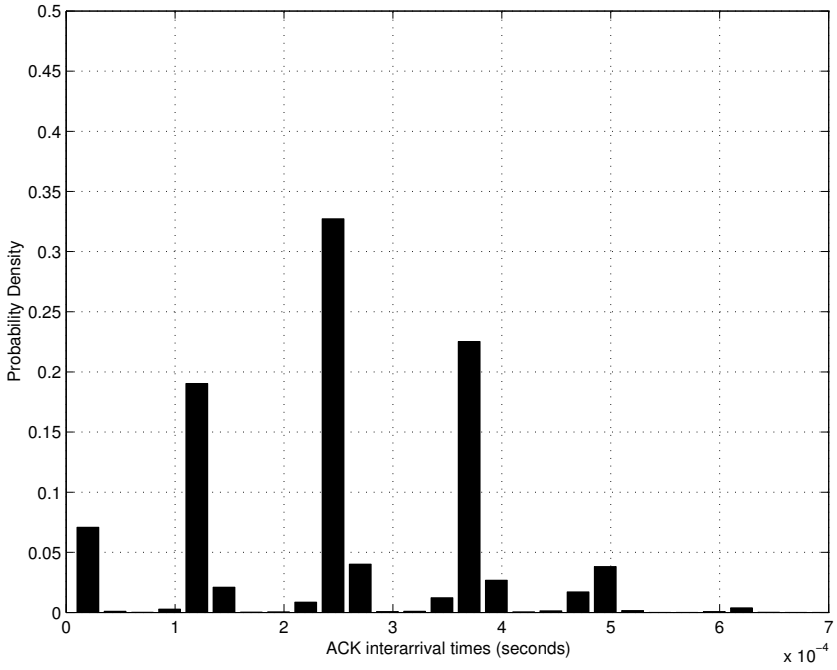

Figure 4.12: ACK inter-arrival distribution at host for 4 active VMs

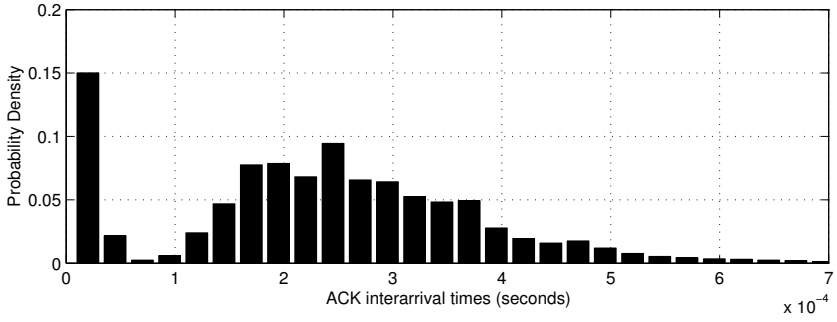

Figure 4.13: ACK inter-arrival distribution at vm1 for 4 active VMs

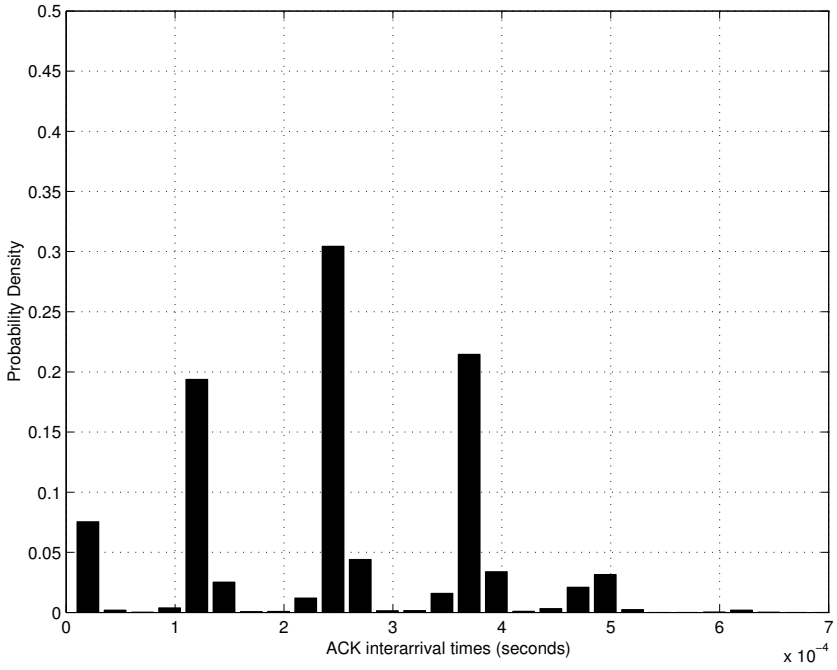

Figure 4.14: ACK inter-arrival distribution at host for 5 active VMs

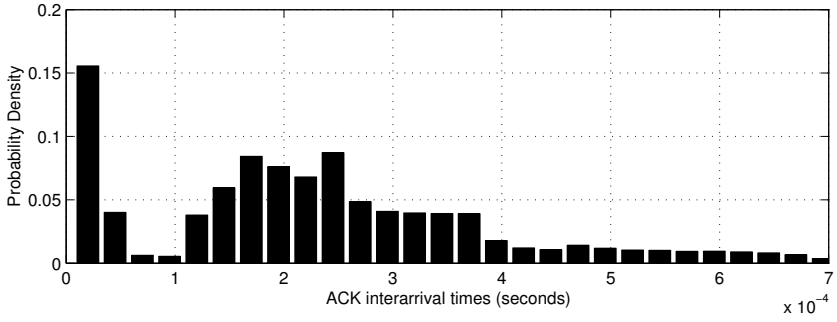

Figure 4.15: ACK inter-arrival distribution at vm1 for 5 active VMs

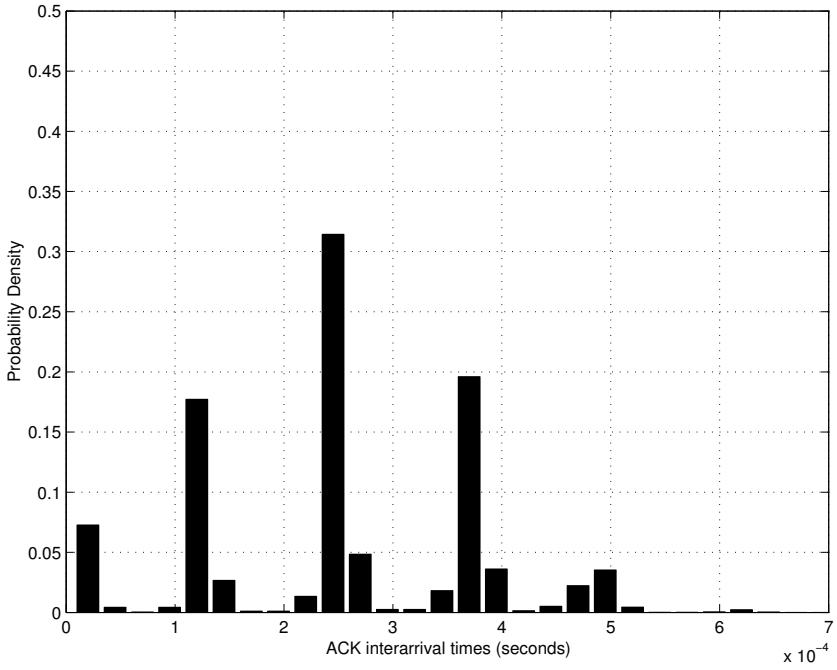

Figure 4.16: ACK inter-arrival distribution at host for 6 active VMs

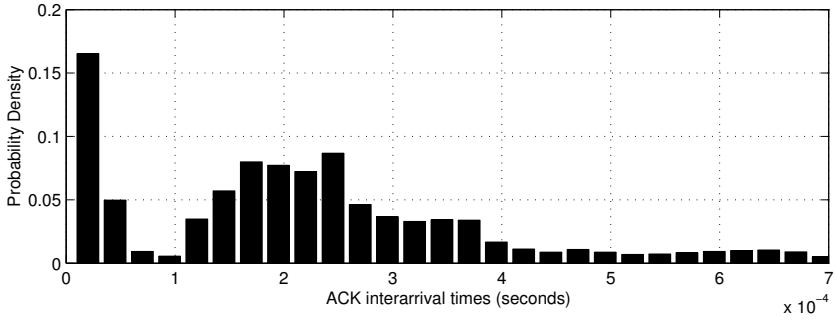

Figure 4.17: ACK inter-arrival distribution at vm1 for 6 active VMs

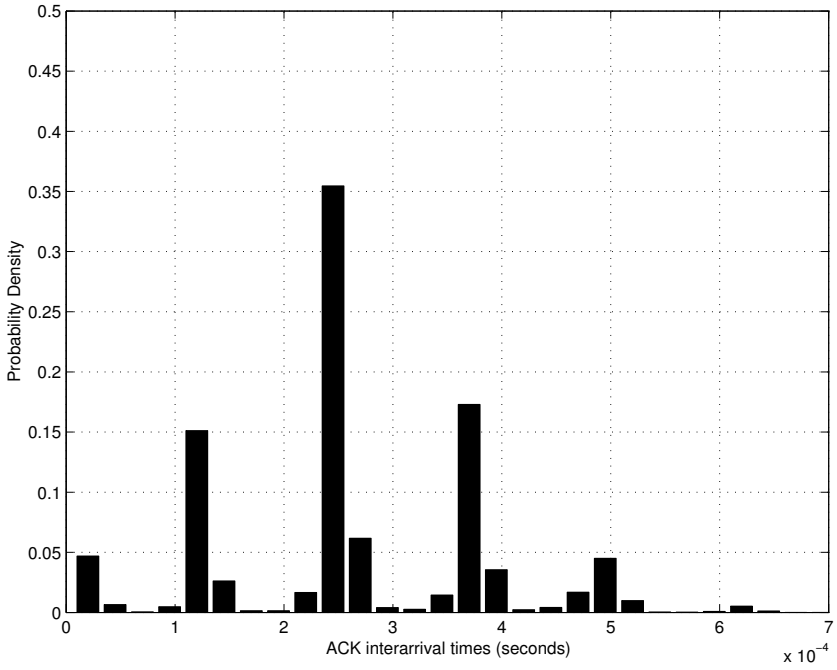

Figure 4.18: ACK inter-arrival distribution at host for 7 active VMs

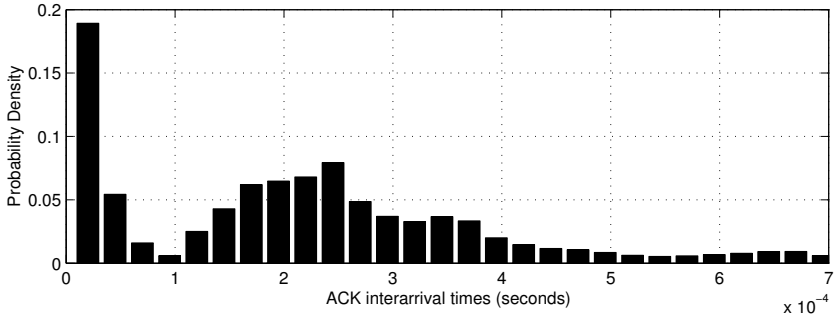

Figure 4.19: ACK inter-arrival distribution at vm1 for 7 active VMs

Next, we empirically show that the level of ACK compression increases with the increase in the number of active virtual machines. Figures 4.8 to 4.19 show inter-arrival distribution of the acknowledgments as seen at the host and vm1 as a function of the

number of concurrent virtual machines. From Figures 4.8 to 4.19, it is clearly evident that the level of ACK-compression and the number of VMs running are highly correlated. Further, the distributions seen at the host are very similar to the baseline suggesting that the ACKs are not compressed in the network. Figure 4.19 shows a large mode centered at 0.045ms. It shows that about 18% of the ACK arrive at vm1 with a spatial gap of 0.045ms when 7 VMs are running as against 15% when 4 VMs (Figure 4.13) are running and 8% when only 1 VM (Figure 4.5) is running. Further, the increase in processing delay is also clearly evident from the figures. The increase in the size of the tail of the inter-arrival distribution in Figure 4.19 indicates the significant increase in the delay caused by the virtualization overhead.

The implications of this observation on the network dynamics are studied in the following chapter.

Chapter 5

Network Side Analysis

In this chapter, we show that under heavy loads, the burstiness and packet loss rate of TCP connections and consequently the mean Web response times increase as the number of VM hosted servers increases.

5.1 Network Configuration

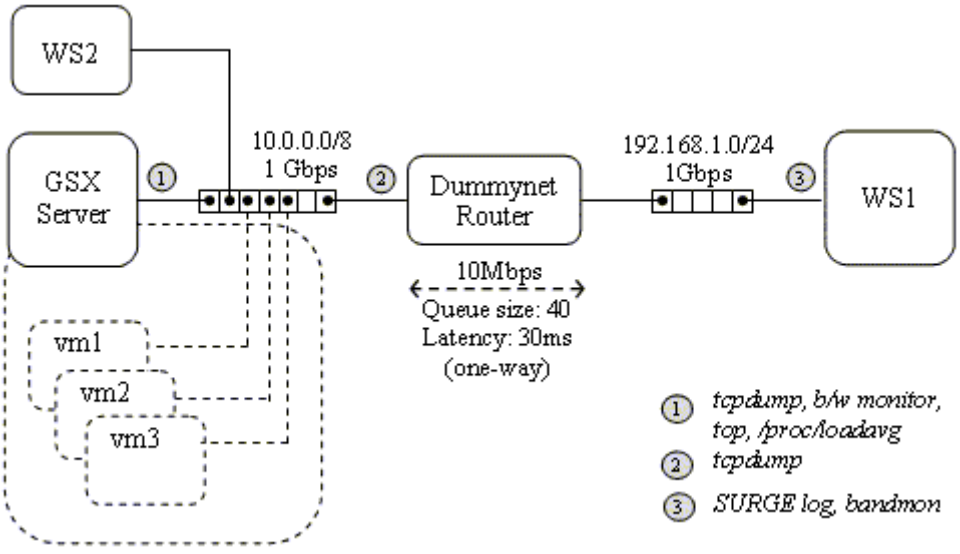

Figure 5.1 Network Configuration for network side analysis

The network configuration used for this analysis is shown in Figure 5.1. *Dummynet* [26] is configured to use two unidirectional pipes, each with a bandwidth of 100 Mbps and a

one-way latency of 30ms. The length of the *dummynet* queues along each direction is set to 40 packets. This setup emulates a typical WAN latency for the data traffic between the two local area networks that the router interconnects.

5.2 Experimental Methodology

In this set of experiments, the impact of VM hosted Web-servers on WAN performance is studied as a function of the traffic load and the number of active virtual machines. The VMware GSX server acts as a farm of Web servers and is subject to varying load levels of http traffic from SURGE. As the number of virtual servers increase, we look for invariant impacts on the network and application performance.

The machines labeled WS1 and WS2 are the source of Web traffic. SURGE software is run on these two machines. For the studies reported here, the number of simulated Web clients on WS1 takes the values {100, 200, 300, 400 and 600} and the number of clients on WS2 takes the values {100, 200, 300, 400 and 300}. The last setting required a large number of users at WS1 than at WS2 to provide increased traffic over the simulated WAN. The overloaded system could not fully consume the 100Mbps WAN bandwidth. During our initial study, we found that SURGE was not able to provide a high load level over long time periods. It stopped with a segmentation fault after roughly 32,000 Web objects have been received from the server. Hence, at higher load levels, we used multiple instances of SURGE each emulating a maximum of 150 Web-clients.

The number of virtual machines used is {0, 2, 4, and 7}. The ‘0-vm’ case serves as the baseline run. The unmodified host operating system on the GSX server is used in

this case. The Apache server on the host is configured with higher *MaxClients* setting of 700 to accommodate the higher aggregate loads [1]. For each of the twenty possible combinations of Web clients and virtual servers, the values reported are obtained from a run of approximately ten minutes of elapsed real time. Five http connections that lasted for roughly about 20 to 30 seconds are picked and analyzed. The system is configured to equally distribute the load among the virtual machines.

5.3 Performance Data

Performance data is collected at multiple locations within the network testbed. The tools and their respective vantage points are shown in Figure 5.1. The metrics of interest are: aggregate network load at the WAN and the server, aggregate http requests-per-second as seen at the network interface of the GSX Server, CPU load as experienced by the GSX Server, loss-rate of individual TCP connections, *Web response times* and *burstiness*. Web response time is defined as amount of time from when a SURGE client issues a request to a Web object to when the object has been successfully received by the client. The burstiness metric is defined as follows: Let n be a constant time interval (60ms in our case), $\{t_i\}$ = the number of bytes acknowledged during the i^{th} constant subinterval n of the TCP connection and μ_t and σ_t represent the mean and standard deviation of the $\{t_i\}$. Burstiness of the session is defined as μ_t/σ_t, the coefficient of variation of the series $\{t_i\}$. All aforesaid metrics are obtained by post-processing the data generated by the monitoring tools.

tcpdump was run at the host interface of the GSX server. A filter was used with this instance of *tcpdump* to capture only the SYN and FIN packets for port 80 (http) seen in the data stream. The dump was post-processed to obtain the aggregate http request rate for the GSX Server. The data on network load is obtained from the */proc* file system of the GSX server and the SURGE client (WS1) similar to the set of experiments described in chapter 4.

CPU utilization on the GSX server is inferred from the load average data obtained from the */proc/loadavg* file of the GSX server and is also monitored in real time with the *top* utility. An instance of *tcpdump* is also run at the 10.0.0.0/8 network interface of the dummynet router. The characteristics of individual TCP sessions such as the loss rate and burstiness are recovered from the trace file generated with this instance of *tcpdump*. Our data collection points are consistent with the methodology suggested by Mogul in [22]. In our analysis, we picked five representative TCP connections per trace and looked at the average of the metrics obtained for those connections. The Web response times are obtained from the SURGE log files.

5.4 Results and Analysis

In this section, we show that under heavy loads, the burstiness and packet loss rate of TCP connections and consequently the mean Web response times increase with the number of VM hosted Web servers as compared to a single server running no VMs and being subject to similar network load. In the process, first we show that the network loads over the WAN produced by different SURGE workload configurations was consistent.

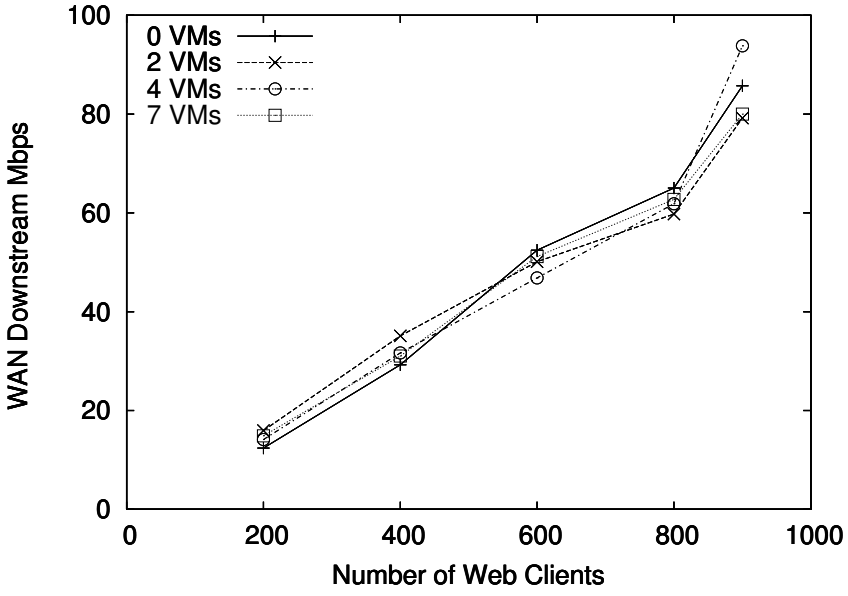

Figure 5.2 WAN Load

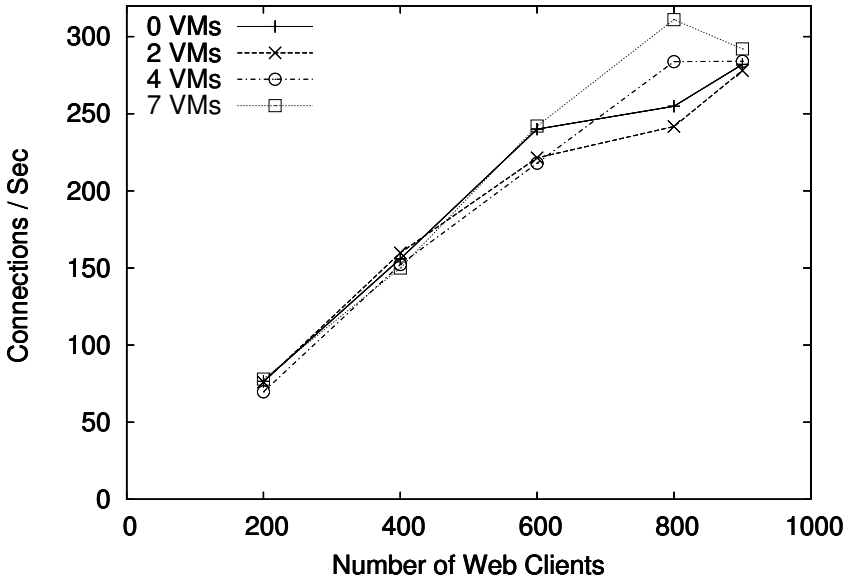

Figure 5.3 *http* Connection Rate

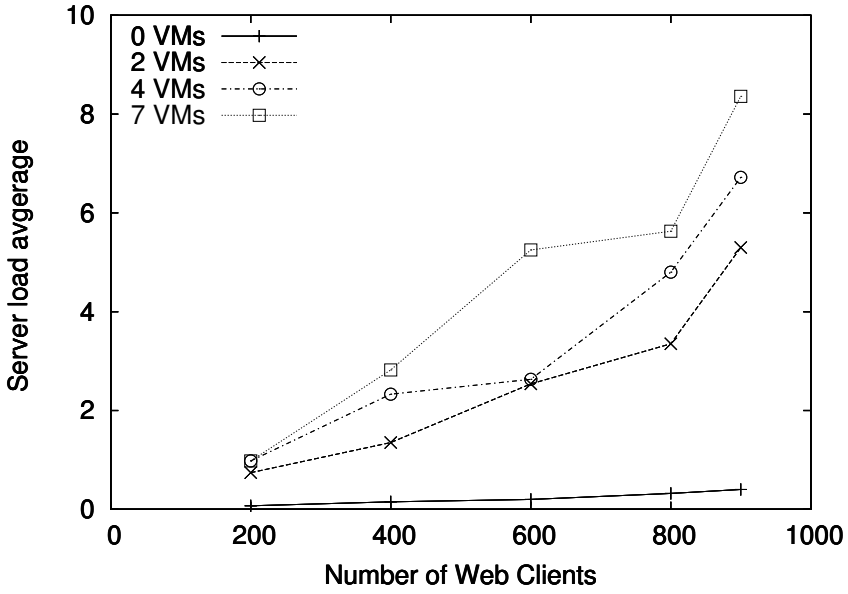

Figure 5.4 Server Load Average

Figure 5.2 shows the aggregate downstream throughput traversing the emulated WAN is shown as a function of the number of SURGE clients. The data shown is the actual arrival rate at WS1. The small, but visible differences in the aggregate throughput are due to the stochastic nature of the SURGE clients. Figure 5.3 shows the http request rate as seen at the host interface of the GSX Server as a function of the number of SURGE clients. These two figures show that the GSX Server was subject to similar network load levels as the number of active VMs was varied as 0, 2, 4 and 7 respectively.

Figure 5.4 shows the utilization of the processor in the host machine as a function of the number of SURGE clients. The data shown is collected from */proc/loadavg* file of the GSX Server. It represents the exponential weighted average of the number of

processes running or ready to run during the previous 5 minutes. It was collected towards the end of each run. Our experiments for each load level lasted for about 10 minutes of elapsed real time. This graph clearly shows the extent of CPU overhead imposed by virtualization of Web servers although the GSX server was subject to identical network load levels. At high loads, the GSX server was highly CPU bound when the virtual machines were running. With 800 total web clients and 7 VMs, the top utility reported 100% utilization of both CPUs. It is important to note that when comparing the baseline against other cases, the system is much busier. This in turn reflects in higher levels of ACK-compression and as we will show it also results in higher burstiness of TCP sessions. As we mentioned in chapter 4, this additional CPU overhead causes the TCP ACK stream to be perturbed in a great way.

Next, we show the implications of the results of chapter 4 on the network dynamics of a TCP connection. We described the negative impacts of ACK-compression in section 2.7. Ideally, for a TCP session in equilibrium with the self-clocking in effect, there are no large variations in the bytes transferred per RTT. However, when the ACKs are compressed it results in a breakdown of the self-clocking mechanism of TCP. The sender is misled into sending more data in a burst than the network can accept, which can result in network stress and packet loss. The uncongested RTT for the testbed was 60ms. Hence, we use 60ms as our constant time-interval(n) in the computation of the burstiness metric associated with a TCP session. In chapter 4, we showed that the level of ACK-compression increased with the increase in the number of active VMs. Based on that we conjectured that the use of VM hosted servers would lead to higher levels of bursty TCP

sending behavior as the load and the number of active virtual servers on the GSX server increased.

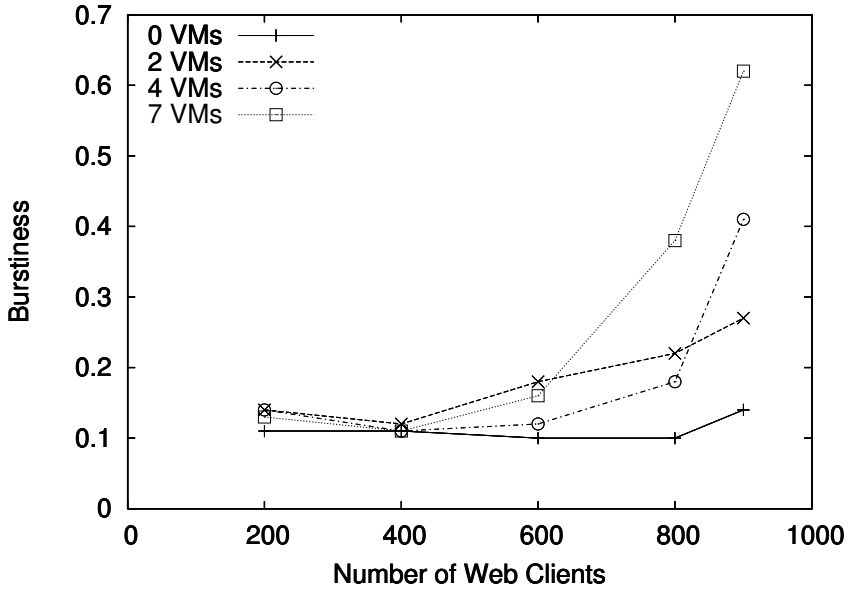

Figure 5.5 Send Burstiness

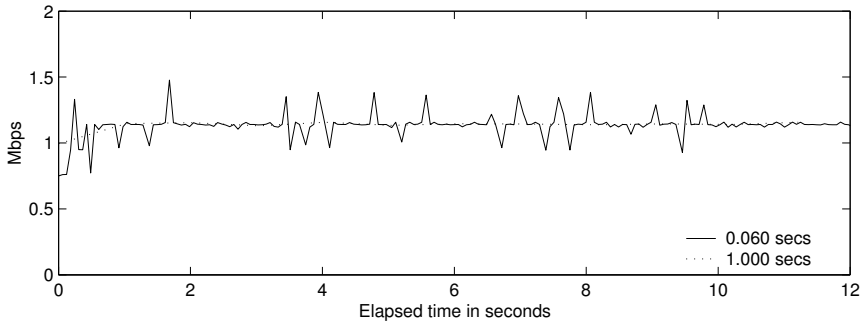

Figure 5.6 Throughput (no VM running)

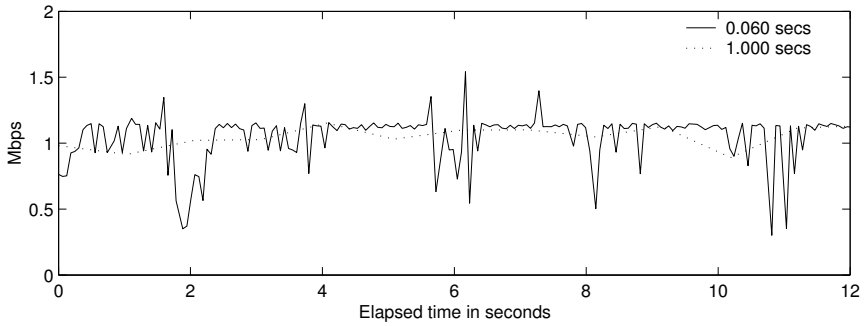

Figure 5.7 Throughput (2 VMs running)

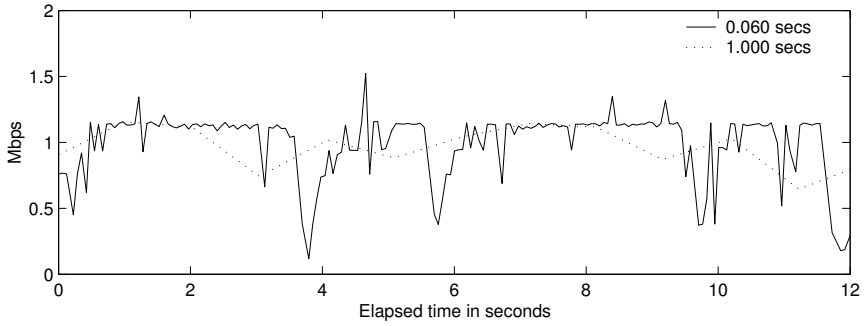

Figure 5.8 Throughput (4 VMs running)

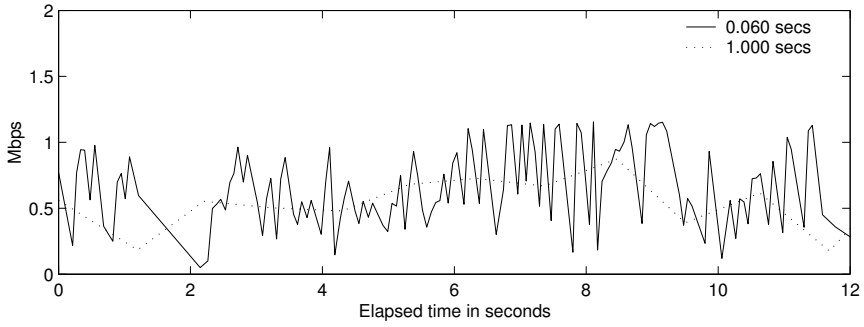

Figure 5.9 Throughput (7 VMs running)

Figure 5.5 show the burstiness metric against various SURGE load levels when different number of VM hosted servers was active. Each point represents the mean value of burstiness metric for the five largest downloads that occurred for the corresponding number of VM hosted servers and load level. From the figure it is clearly evident that the burstiness associated with a TCP session increases with the increase in the number of VM hosted servers.

Figures 5.6, 5.7, 5.8 and 5.9 show the observed throughput of individual TCP sessions with 0, 2, 4, and 7 VM hosted servers respectively for a SURGE load level of 800 Web clients. These graphs illustrate the burstiness in an alternate way. The line labeled 0.060secs shows throughput in Mbps as the number of bytes acknowledged during every 60ms interval of the TCP session under observation. The line labeled 1.000secs shows the observed throughput in Mbps over a 1 second timescale. As noted earlier from Figure 5.5 the burstiness of a TCP connection when no VM hosted servers are used is lower compared to that of VM hosted servers. Consequently, we do not see much variation in the short-term throughput in Figure 5.6. The short-term throughput converges to a steady state throughput rapidly and exhibits only minor excursions thereafter. The connection consumes a 1Mbps sustained throughput. However, when VM hosted servers are used, we see more burstiness in the short-term throughput. Further, it is clearly evident from Figures 5.7 5.8 and 5.9 that the burstiness metric and the number of active VMs are directly proportional. The dips that we see in the short-term throughput as well the long-term throughput in Figures 5.8 and 5.9 indicate loss. On a smaller

timescale on the order of milliseconds, we observed bursts in excess of 200 Mbps when 7 VMs were active.

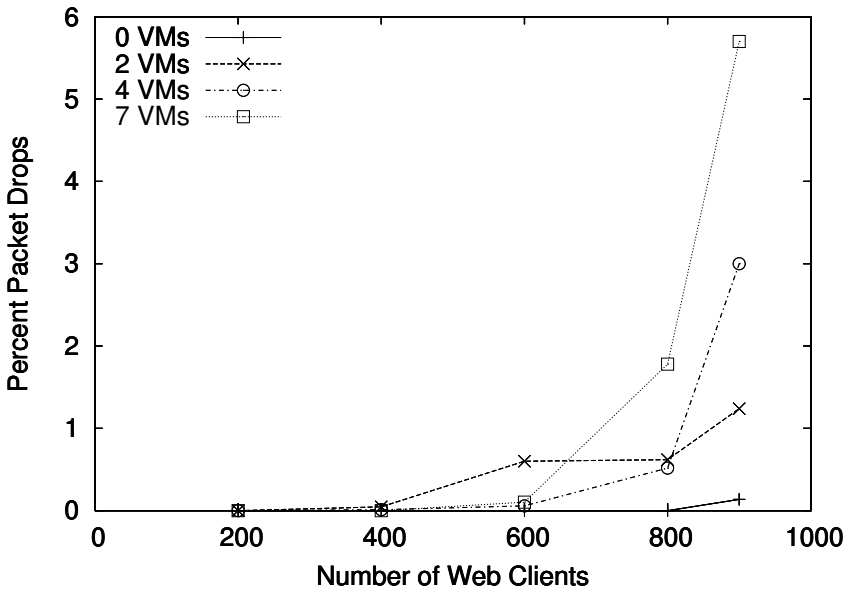

Figure 5.10 Load versus Loss

Figure 5.10 shows the packet loss rates that the individual TCP connections experience at the bottleneck link as a function of the number of SURGE clients. As previous studies have shown the loss rates increase with increase in burstiness. It is clearly evident from Figure 5.10 that under high loads, the loss rates increase with the number of VM hosted servers. A loss rate of 5.7% produced by 7 VM severs at a load of 80 Mbps is more than 40 times the loss rate of 0.13 obtained with no VM hosted servers

at a higher load of 86 Mbps. Figure 5.11 shows the impact of load and the number of active VMs on end-to-end performance by showing the average SURGE Web-response time as the load increases.

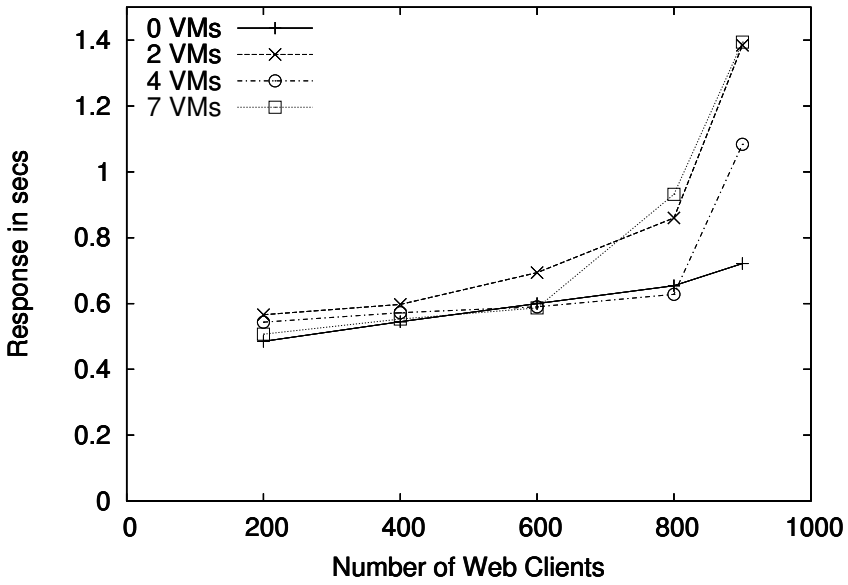

Figure 5.11 Load versus Response time

Thus, we have clearly shown that when a VM based system is subject to heavy loads, the burstiness and packet loss rate of TCP connections and consequently the mean Web response times increased as a function of the number of VM hosted servers.

Chapter 6

Conclusions and Future Work

In this thesis, we have shown that under heavy loads, as the number of active VMs increase, network performance can deteriorate significantly as compared to a non-VM system subject to the same load. Our method involved running experiments on two systems: a VM-based system and a non-VM based system. When subject to identical loads, we show that the VM system can result in significantly worse end-to-end performance than the non-VM system

We have shown that the virtualization overhead greatly affects the ACK stream destined for a virtual machine. We have empirically shown that the level of ACK-compression is directly proportional to the number of active virtual machines. It is well known that ACK-compression can lead to poor TCP performance. We studied the effect of ACK-compression on the network dynamics using a realistic scenario involving a farm of Web servers. We have shown that the use of VM hosted Web servers can negatively affect TCP performance in a measurable way. The burstiness associated with the TCP transmissions from the VM hosted Web servers increased as the number of servers increased.

We have conjectured that this result is due to ACK packets collecting at a queuing point within the VM system while waiting for the VM to be dispatched. It is also possible that optimizations like receive data combining are implemented. As we did not have access to VMware's source code, we are not able to confirm this. It is clear however that

these effects become extreme as the number of active VMs grows. At high loads with multiple VMs running, it takes longer for the target VM to get scheduled. In such a scenario, even in the absence of deliberate delays caused by optimizations in the VMM, more ACKs get queued at the host due to scheduling latencies. As a result, the level of ACK-compression increases which leads to burstier TCP send behavior.

A modern operating system subject to very high loads is also susceptible to large processing delays and hence might induce ACK-compression. However, for comparable network loads on a system with no VM running and another with one or more VMs running, the impact that the system overhead would have on TCP is very different. The processing delays in the no-VM setup are smaller compared to the VM delays. The VM delays are much larger due to the overhead associated with the world switches involved in the functioning of a VM. Hence, for comparable network loads, the two setups mentioned above would exhibit very different system dynamics that impact the network differently.

In our testbed, we saw increased loss rates at the intermediate router. By increasing the queue depth in the router we might have been able to avoid loss. But we have shown an invariant result: a VMware GSX Server can induce bursty send behavior in a VM which is likely to negatively impact network performance. Although, we haven't shown it, we believe this result is applicable to at least all Type II VMMs.

We found another interesting result which actually complicated our analysis. As we have shown, on a busy system, the VM overhead can compress the incoming events' timing before the VM sees the events. At the same time, it is also possible for the inter-

event timing to get delayed or decompressed due to the VM overhead. The VM might see the events occur at a slower and more variable rate compared to that observed at the VM's host machine. This results in a smoothing effect which was apparent when comparing inbound and outbound data streams at the GSX Server and at the VM. This actually complicated our analysis because even though we observed very high levels of ACK-compression at the VM, the subsequent burstiness associated with the outbound data stream was smoothed out when observed at the host.

A future direction for the project is to examine an open source VMM like User Mode Linux (UML) or Xen. UML is a Type II VMM and is implemented using the system call interface provided by Linux. The authors of [10] shown that UML implementation involves several performance bottlenecks. For example, every system call performed by a VM involves switching from the VM process to the tracing thread that is part of the UML and back twice, for a total of four context switches. This is similar to the overhead caused by the world switches in VMware. Further, the block device drivers in UML are able to have only one outstanding I/O request at a time which would greatly affect the performance of I/O intensive applications. Based on these results, we conjecture that the overall performance of VMware would be better than UML due to the various optimizations. With respect to the impact on the ACK stream, we conjecture that we would see some similar results due to the processing that are inherent in any VM system.

Xen is a Type I VMM. It uses *para-virtualization*, wherein the drawbacks of full virtualization are avoided by providing a VM abstraction that is similar but not identical

to the underlying hardware [5]. Thus, unlike VMware, it does not fully virtualize the x86 system for the VM. For instance, VMware emulates the x86 memory management unit by dynamically rewriting portions of the hosted machine code to insert traps wherever VMM intervention might be required [5]. As noted in chapter 2, an Intel's x86 architecture is not naturally virtualizable. An attempt to fully virtualize the x86 architecture reflects in an increased penalty on the performance of the VMM system. More state information is maintained and it becomes a heavyweight operation to context switch between virtual machines. Hence in comparison to the VMware, we would expect to see better performance with Xen. However, the inherent virtualization overhead should reproduce the results that we inferred from our study when the system is subject to heavy load levels although the impact could be less.

As the immediate step ahead, we suggest that we validate our conjecture that all Type II VMMs will show similar behavior be validated using UML and Xen. Following that, one can come up with a metric that gives us the ideal number of virtual machines that could be run for an estimated load level without greatly affecting the network dynamics. In order to alleviate the negative impact as observed from our work, we suggest a bandwidth shaper be implemented in the device driver that operates the physical NIC of the VM system. The bandwidth shaper should be aware of the number of virtual machines running at any given point of time. It does a flow level TCP bandwidth estimate from the ACK stream seen at the host. The outgoing traffic from each virtual machine can be shaped based on this estimate. But, this involves an overhead in addition to virtualization. As well, the state has to be maintained for every TCP session. This

would require more memory. However, memory is cheap and so maybe the bigger challenge is the additional overhead. In a commercial setup that uses a VM based server hosting, the cost savings achieved by using a VMM based system compared to dedicated physical servers are quite substantial. Hence high-end hardware with abundant memory can be used. In such a case, the benefits of running a bandwidth shaper might outweigh the penalty.

References

[1] Apache, http://httpd.apache.org

[2] A Awadallah, M. Rosenblum, "The vMatrix : A Network of Virtual Machine Monitors for Dynamic Content Distribution", 7th International Workshop on Web Content Caching and Distribution (WCW), August 2002.

[3] H. Balakrishnan, et. Al., "TCP Behavior of a Busy Internet Server: Analysis and Improvements", IEEE INFOCOM98, 1998.

[4] P. Barford, M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", Proceedings of Performance, ACM SIGMETRICS98, 1998.

[5] P.Barham, et. Al, "Xen and the Art of Virtualization", 19th ACM Symposium on Operating Systems Principles, October 2003.

[6] Bochs, http://bochs.sourceforge.net

[7] K. Buchacker, V. Sieh, "Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects", Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering (HASE), pp 95-105, Oct 2001.

[8] E. Bugnion, S. Devine, K. Govil, M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors", ACM Transactions on Computer Systems, 15(4):412-447, November 1997.

[9] J.S. Chase, et. Al, "Dynamic Virtual Clusters in a Grid Site Manager", 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), June 2003.

[10] J. Dike, 'A User-mode Port of the Linux Kernel', Proceedings of the USENIX Annual Linux Showcase and Conference, Atlanta, GA. Oct 2000.

[11] G. Dunlap, S. King, S. Cinar, M. Basrai, P. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay", Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI), December 2002.

[12] R. Figuiredo, P. Dinda, J. Fortes, "A Case for Grid Computing On Virtual Machines", Proceedings of International Conference on Distributed Computing Systems (ICDCS), May 2003.

[13] freeVSD, http://www.freevsd.org.

[14] R. Goldberg, "Architectural Principles for Virtual Computer Systems", PhD Thesis, Havard University, 1972.

[15] R. Goldberg, "Survey of Virtual Machine Research", IEEE Computer, pp. 34-45, June 1974.

[16] Intel Corporation, Intel Architecture Developer's Manual, Volumes I, II and III, 1998.

[17] V. Jacobson, "Congestion Avoidance and Control", ACM SIGCOM88, Aug 1988.

[18] X. Jiang, D. Xu, “SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms”, Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, WA, June 2003.

[19] S. King, G. Dunlap, P. Chen, “Operating System Support for Virtual Machines”, Proceedings of the 2003 USENIX Technical Conference, June 2003.

[20] P. Magnusson, B. Werner, “Efficient Memory Simulation in SimICS”, Proceedings of the 1995 Annual Simulation Symposium, April 1995.

[21] W.McEwan, “Virtual Machine Technologies and their Application in the Delivery of ICT”, Proceedings of the 15th Annual, NACCQ, July 2002.

[22] J. Mogul, “Observing TCP Dynamics in Real Networks”, Technical Report, Digital Western Lab, April 1992.

[23] V. Paxson, “Measurements and Analysis of end-to-end Internet Dynamics”, Ph.D. dissertation, Univ. California, Berkeley, CA, 1997.

[24] Plex86, http://plex86.sourceforge.net/

[25] G.Popek,R. Goldberg, "Formal Requirements for Virtualizable 3rd Generation Architectures", Communications of the A.C.M, 1974.

[26] L. Rizzo, “Dummynet: A Simple Approach to the Evaluation of Network Protocols”, Computer Communication Review, 27(2), Feb 1997.

[27] J.S. Robin, C.E. Irvine, “Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor”, Proceedings of the 9th USENIX Security Symposium, August, 2000.

[28] P. Sarolahti,A. Kuznetsov, "Congestion Control in Linux TCP", Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, 2002.

[29] SimOS, http://simos.stanford.edu/ [SIMOS]

[30] J.E. Smith, “An Overview of Virtual Machine Architectures”, Oct 2001.

[31] J. Sugerman, G. Venkitachalam, B. Lim, “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor”, Proceedings of the USENIX Technical Conference, June, 2001.

[32] tcpdump, http://www.tcpdump.org

[33] Virtual PC, http://www.microsoft.com/windowsxp/virtualpc

[34] VMware, http://www.vmware.com.

[35] J.M. Westall, tcpmon – packet filter for Linux kernel 2.4

[36] A. Whitaker, M. Shaw, S. Gribble, “: Lightweight Virtual Machines for Distributed and Networked Applications”, Proceedings of the USENIX Annual Technical Conference, Monterey, CA, June 2002.

APPENDIX A

Script for bandwidth monitor

```
/*********************************************************************
* Simple application that reports the bandwidth utilization at
* an network interface
*
* To change the frequency of probing, change the REFRESH macro below
*********************************************************************/

#include <string>
#include <sys/time.h>
#include <unistd.h>
#include <iostream>
#include "dev.h"

#define REFRESH   2000000 // microseconds
using std::string;
using std::cout;
using std::endl;

int main(int argc, char ** argv)
{
    if( argc != 2 ){
      printf("\n Usage: band_mon <interface name> \n \t\t
                  eg: band_mon eth0\n");
      exit(2);
    }
    Device dev1(argv[1]);
    struct timeval cur_time;
    unsigned long data[4];
    int status = dev1.getStat(data) ;

    while(status>=0){
                gettimeofday(&cur_time, NULL);
                cout << (unsigned long)(cur_time.tv_sec) <<":"
                      << (unsigned long)(cur_time.tv_usec) << "  "
                      << data[0]<< "  "
                      << data[1]<< "  "
                      << data[2]<< "  "
                      << data[3]<< "  "
                      << endl;
                usleep(REFRESH);
                status = dev1.getStat(data) ;
    }
    if(status == -1){
      printf("\n Error: reading /proc/net/dev \n");
      return 1;
    }
    if(status == -2){
```

```
      printf("\n Error: the device does not exist or is currently
down\n");
      return 2;
    }
}

/* The Device class
*/

#include <string>
#include <cstring>
#include <stdio.h>

using std::string;

class Device {
      private:
            unsigned long last[4];
            string devName;
            Device(){}

      public:
            Device(char* t): devName(t) {;}
            int getStat(unsigned long current[4]);
};


int Device::getStat(unsigned long current[4]){
      int ret_val =-1;
      FILE *fd;
      char buf[512] = "";
      char tag[128] = "";
      char *tmp, *tmp2;

      unsigned long total_new[4];

      if( ( fd = fopen( "/proc/net/dev", "r" ) ) == NULL )
            return ret_val;

      ret_val=0; //no change in values - must remove: not used now

      fgets( buf, 512, fd );
      fgets( buf, 512, fd );

      while( !feof( fd ) )
      {
            fgets( buf, 512, fd );
            memset( tag, 0, 32 );
            tmp = buf;
            tmp2 = tag;

            while( *tmp == ' ' ) tmp++;
            while( ( *tmp2++ = *tmp++ ) != ':' );
```

```
            *--tmp2 = '\0';

            float d;
           sscanf( tmp, "%u %u %f %f %f %f %f %f %u %u", &total_new[0],
&total_new[1], &d, &d, &d, &d, &d, &d, &total_new[2], &total_new[3] );

            if( ! strcmp( devName.c_str(), tag ) )
            {
                  current[0] = total_new[0];
                  current[1] = total_new[1];
                  current[2] = total_new[2];
                  current[3] = total_new[3];

                  ret_val=1; // value has changed
                  fclose( fd );
                  return ret_val;
            }
      }
      ret_val=-2;

      fclose(fd);
      return ret_val;
}
```

APPENDIX B

Script for *dummynet* configuration

```
#-----------------------------------------------
#
# 192.168.1.0/24 <----------------> 10.0.0.0/8
#                  rtt:60ms bw:100Mbps
#
#-----------------------------------------------

ipfw add pipe 1 ip from 192.168.1.0/24 to 10.0.0.0/8 out xmit em0
ipfw add pipe 2 ip from 10.0.0.0/8 to 192.168.1.0/24 in recv em0
ipfw pipe 1 config bw 100Mbits/sec queue 40 delay 30
ipfw pipe 2 config bw 100Mbits/sec queue 40 delay 30
```