ON THE IMPACT OF VIRTUAL MACHINE OVERHEAD ON TCP PERFORMANCE

V. Rajasekaran, J. Martin and J. Westall
Department of Computer Science
Clemson University
Clemson, SC 29634-0974
Email: vrajase, jmarty, westall@cs.clemson.edu

ABSTRACT

Virtual Machine (VM) based systems are widely used in the Internet and in corporate networks. In previous work we presented evidence that the virtualization overhead in a VM system perturbs TCP connections such that network performance deteriorates as compared to a non-VM system subject to the same user load. In this paper, we provide an explanation for these results. We show that that the underlying cause is due to VM overhead which induces ACK-compression resulting in highly bursty send behavior by a VM serving as a TCP end-point.

KEY WORDS
Internetworking, Network Performance, Virtual Machines

1. Introduction

The virtual machine was originally conceived by IBM in the 60's for their IBM System 360 mainframe class of computers [1]. It was an alternative approach to the conventional time-sharing operating systems. The structure of a traditional virtual machine system is shown in Figure 1. The Virtual Machine Monitor (VMM) is a software that provides users the appearance of direct access to a dedicated machine [2]. It emulates multiple instances of the underlying physical hardware platform. The environment created by the VMM is called a Virtual Machine (VM). The virtual machine provides an illusion of exclusive access of the computer hardware on a per user basis. In this way, a VMM can efficiently multiplex the physical hardware across several instances of one or more operating systems running concurrently.

1.1 Applications of VM Systems

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. Industry trends and technology including server consolidation, shared-memory multiprocessing and low cost PC based distributed computing have led to a renewed interest in VM technology. Virtual Machine solutions for PC platforms such as VMware [3], Virtual PC [4], and User Mode Linux [5] can provide significant cost savings to corporations. They allow under-utilized servers to be virtualized onto smaller number of physical machines thereby reducing support and management costs [6].

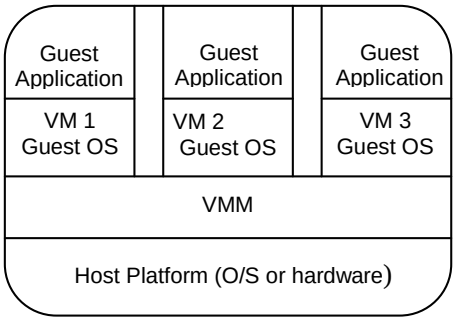

Figure 1: VM Architecture

1.2 Performance of VM Systems

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 [1]. Further, there are several technical and pragmatic hurdles, which can impose serious performance obstacles that arise when virtualizing the PC platform [2, 7]. These performance obstacles motivated a number of studies whose objective is to reduce CPU overhead in a VM environment. Recent performance studies have found that on modern hardware, and with certain performance optimizations, VM overhead can be limited to less than 50% [7, 8, 9]

The majority of related research has focused on the internals of VMM systems. To the best of our knowledge, little or 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 important to understand the impact that these systems can have on the Internet.

In previous work, we studied the impact that a VM system has on network performance using a realistic scenario involving a farm of VM Web servers [10]. We showed that under heavy usage loads and with a moderate

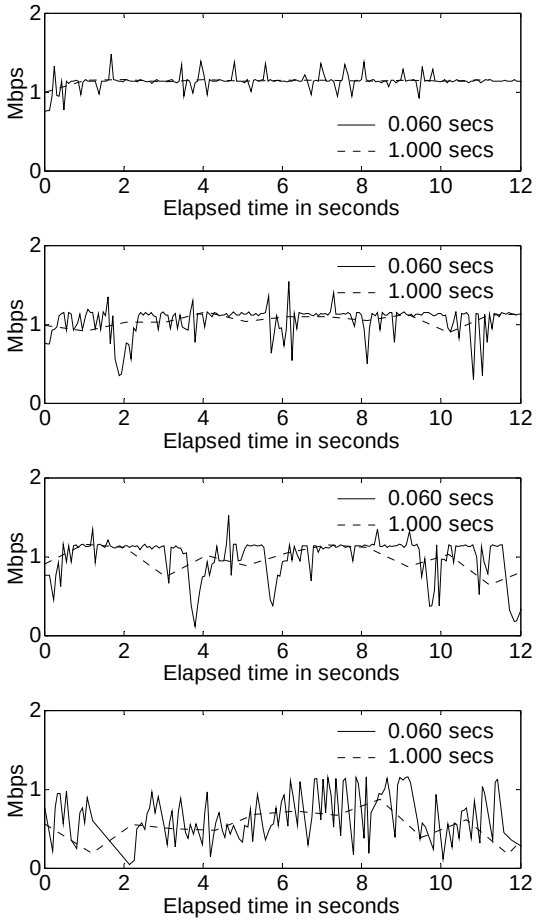

Figure 2: Single Session Throughput

number of active VMs running, network performance can deteriorate significantly as compared to a non-VM system subject to the same user load. The testbed for the study involved a set of Web servers running on VMs created using VMware's GSX Server, a *dummynet* [11] emulated WAN configured for a 100Mbps link with a 60ms RTT, and the SURGE [12] tool to generate realistic Web loads. We showed that under heavy loads the TCP loss rates and consequently the mean response time of http transactions increase with the number of VM hosted Web servers. The same machine configured without VMs and subject to the same Web user load suffered almost no loss and achieved higher utilization of the WAN. Figures 2a to 2d illustrate this result by showing the throughput observed at the intermediate router for individual long-running TCP sessions with 0, 2, 4 and 7 VM hosted servers respectively, when subject to the same amount of Web traffic load (800 SURGE users). The line labeled 0.060secs shows throughput in Mbps as the number of bytes acknowledged during every 60ms interval (RTT in our case) of the TCP session under observation. The line labeled 1.000secs shows the observed throughput in Mbps over a 1 second timescale. The first of these graphs shows that with no VM hosted servers the short-term throughput converges rapidly to the long-term throughput and exhibits only minor excursions thereafter. All of the other graphs show continuing excursions from the mean. The frequency of the excursions, which characterizes the burstiness of the TCP connections, also increases with the number of VM hosted servers.

In this paper, we provide an intuitive explanation of the observed behavior. Throughout this paper, we refer the physical machine hosting the VMs as the VM host and the guest OS as the VM. In [10] we conjectured that the ACK stream arriving at the physical NIC of the VM host gets compressed by the time it reaches the virtual network interface of the VM which, leads to bursty TCP transmissions. In this paper we validate this conjecture. 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 can become extreme as the number of active virtual machines increase.

The remainder of this paper is organized as follows. In section II we provide an overview of the relevant background concepts. In section III we describe the network testbed and the experimental methodology. The results of our study are presented in section IV and conclusions are given in section V.

2. Background

2.1 VMware's Hosted VM Architecture

The VMware's GSX Server version 2.5.1 [3] provides the virtual machine environment employed in this research. It is based on VMware's hosted virtual machine architecture. This architecture requires an underlying *host* OS (Figure 1), and the VMM relies upon that *host* OS for device support [7]. The OS running on the VM is referred as the *guest* OS and the applications that they run are referred to as guest applications. Similar to the process context switches, the authors of [7] define a *world switch*. In this VM architecture, at any point in time the physical processor is executing either in the *host world* (host OS) or the *VMM world* (VMM software) [7]. The context switch between the host world and the VMM world is called a world switch and it involves saving and restoring all user and system visible state on the CPU. It is more heavy-weight than a normal process context switch. When a *guest* application performs an I/O operation such as reading a file, or transmitting a packet across the physical network, the VMM initiates a world switch rather than accessing the physical hardware directly. Following the switch to the host world, the requested I/O operation is performed on behalf of the VM through appropriate system calls on the host OS. 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. Thus the functioning of a virtual I/O device such as the network card involves a large number of world switches leading to very high CPU overhead compared to the functioning of an un-virtualized I/O device. The authors of [7] have shown that the majority of the virtualization overhead is

due to the high number of world switches. They proposed an optimization called *send-combining*, which delays the transmission of outbound packets reducing the number of context switches between the guest OS, the VMM and the host OS [7].

2.2 ACK-Compression

For TCP connections 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. The TCP sender self-clocks itself so that the sending rate never exceeds the bottleneck link speed [13]. 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. Paxson [14] 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. 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 [14, 15, 16].

3. Network Testbed and Experimental Methodology

3.1 Network Configuration

The network testbed that we used for our experimentation is shown in Figure 3. 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. The machines WS1 and WS2 are identical Dell Optilex GX250 PCs equipped with 512 MBytes of RAM and an Intel Pentium IV processor running at 2.4GHz. WS1 runs FreeBSD 4.8. WS2 runs RedHat 8.0 Linux distribution with kernel 2.4.20.

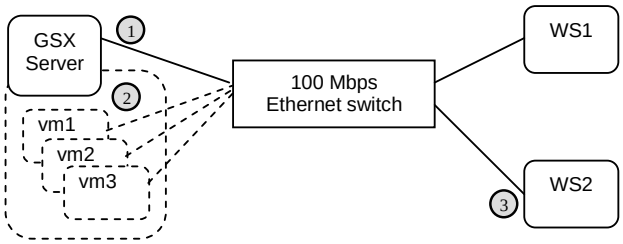

Figure 3: Network Testbed. The various data collection tools used are: 1) tcpdump, b/w monitor 2) tcpmon 3) tcpdump

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 [7, 8, 9] allowed us to calibrate our methods. The host operating system is RedHat 8.0 Linux distribution with SMP kernel 2.4.20. Seven virtual machines were created using the VMware GSX server. Each VM is configured with 256 Mbytes of RAM, 4 GBytes virtual hard disk, and one virtual Ethernet network adapter. The GSX server is configured to run bridged networking [3] for all the virtual machines in which each VM appears as a physical machine present on the LAN. The guest operating system is RedHat 8.0 Linux distribution with kernel 2.4.20.

A simple TCP data transfer program is used to generate the workload for our experiments. 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.

3.2 Experimental Methods

The study is conducted by varying the number of active VMs while subjecting the GSX server to a consistent network load. By varying the number of active VMs, we vary the levels of virtualization overhead. The higher the number of the active VMs, greater is the number of context switches among VMs and hence larger virtualization overhead. 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 VM is started. The virtual machine labeled vm1 is the test machine for all the runs. A single TCP connection is established between vm1 and WS2. An application on vm1 sends about 150 Mbytes to WS2 resulting in roughly 50,000 ACKs arriving inbound at the GSX Server (VM host) destined for vm1. WS1 is used to generate background traffic for the other VMs when multiple VMs are active. The background traffic was limited to a single TCP connection per VM that was similar to the test connection. The ACK interarrival distribution at vm1 is compared against the ACK inter-arrival distribution at the VM host to analyze the impact of virtualization overhead on the ACK distribution. Increase in the number of active VMs results in increased

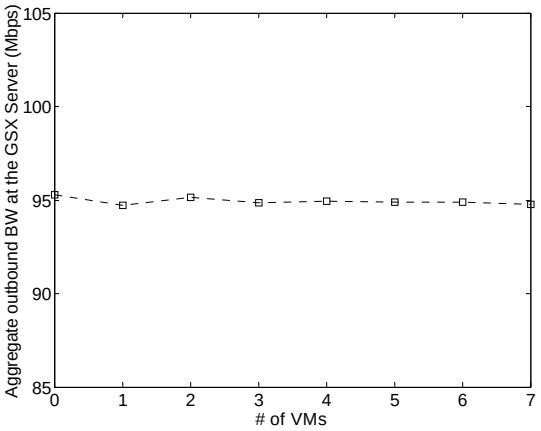

Figure 4: Test Workload

contention for the network card and subsequently higher number of world switches. The aggregate network load, which is measured as the aggregate outbound bandwidth at the GSX Server is kept the same for all the runs.

3.3 Performance data

Performance data is collected at multiple locations within the network. The various vantage points of the tools are shown in Figure 3. At vm1, a customized kernel module called *tcpmon* is used to capture all the data packets of our test connection between vm1 and WS2. 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 [17] using the RDTSC instruction. This data is post-processed to obtain the ACK interarrival times.

The correctness of our analysis rests on the accuracy of the timing data reported by the *tcpmon* program. Initially, we used *tcpdump* 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 smaller 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 [7] have used a similar methodology to obtain the timing information in their work with VMware Workstation.

The ACK stream arriving at the VM host is captured by *tcpdump*. 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. 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. We also show that the level of ACK-compression increases as the number of active VMs increase.

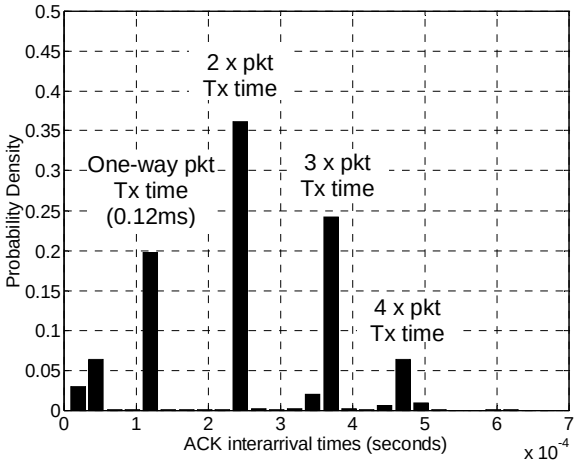

Figure 5: ACK interarrival distribution for the baseline test (no VM)

Figure 4 shows that the amount of traffic generated by the GSX Server and the VMs were the same in all experiments. The data shown is the actual departure rate at the GSX Server. Roughly 95 Mbps of sustained bandwidth was consumed saturating the 100 Mbps network. We can see that the network load remained the same irrespective of the number of virtual machines running concurrently.

In our setup, the two TCP end-points 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. Figure 5 illustrates that we observed this behavior. However, our focus is on the ACK-compression that occurs at the VM system, which, as we show, can induce much higher levels of compression.

Figure 5 shows the ACK interarrival distribution as seen at the host machine's physical NIC for our baseline run. From the figure we see that the interarrival 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 configured for the TCP delayed ACK option. This explains the largest mode (37%) seen at an ACK interarrival time of 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 [18] and 2) possible timing variations in the system. Roughly 8.9% of the ACKs arrive with a spacing less than 0.12ms interarrival time. This suggests ACK-compression. We conjecture that this is caused by receiver compression that occurs within WS2. The other modes represent higher numbers of data segments being acknowledged per ACK. These are caused by ACKs being dropped and/or system delays at WS2.

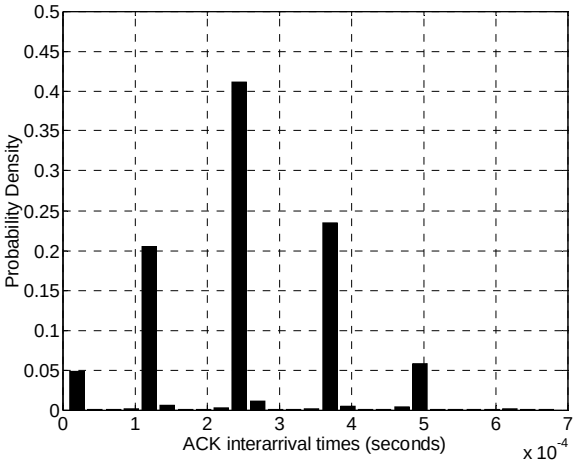

Figure 6: ACK inter-arrival distribution for 1 active VM at the VM Host

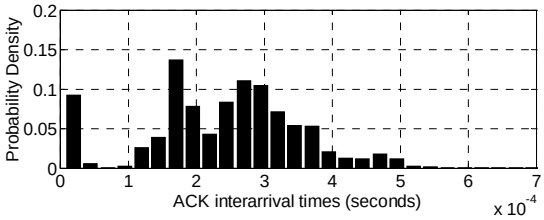

Figure 7: ACK inter-arrival distribution at vm1 for 1 active VM

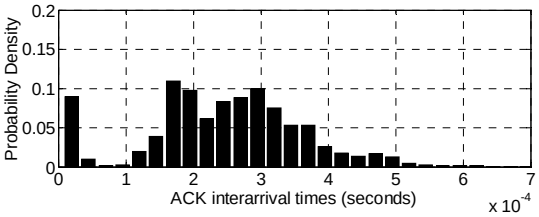

Figure 8: ACK inter-arrival distribution at vm1 for 2 active VMs

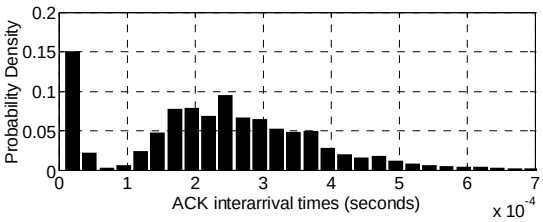

Figure 9: ACK inter-arrival distribution at vm1 for 4 active VMs

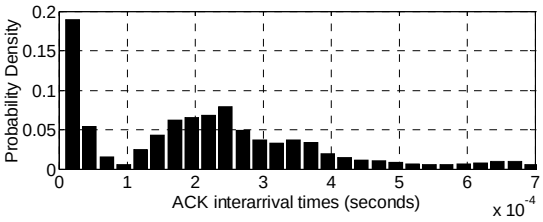

Figure 10: ACK inter-arrival distribution at vm1 for 7 active VMs

Next, we show that the virtualization overhead has considerable impact on the ACK stream by comparing our baseline run and the run with only one active VM. Figures 6 and 7 show the ACK interarrival distribution as seen at the VM host and at vm1 respectively when only one VM is active. The distribution shown in Figure 6 is very similar to that of the baseline (Figure 5). This suggests the absence of other sources of compression in the network. In the absence of VM effects, we would expect the ACK interarrival distribution seen at vm1 (Figure 7) for the 1vm case to be similar to the distribution observed at the host (Figure 6). However, Figures 6 and 7 show that the ACK arrival distribution is greatly perturbed.

Figure 7 suggests that the ACK stream is affected in two different ways: 1) a significant number of ACK arrivals are compressed 2) the ACK interarrival distribution is smoothed out. ACK-compression can be observed by looking at the modes for interarrival times in the sub 0.12ms window of the 1 vm case. Only 5% of the ACKs arrive have an interarrival time less than 0.12ms when they reach the host machine. However when the same ACK stream reaches the VM, about 11% of the ACKs have a spatial gap less than 0.12ms 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 6 have been greatly reduced when the same ACK stream is observed at the VM Figure 7.

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. Under heavy loads with multiple competing VMs the dispatch delay can be significant. The ACKs get queued up at the host and are delivered in a burst when the target virtual machine is eventually dispatched. Consequently we would expect an increased level of ACK-compression when more VMs are active. The results shown later in this section corroborates this conjecture. Secondly, it is possible for an VM implementation to purposely delay delivering ACKs to the VM in hopes of additional network arrivals, similar in concept to *send-combining* in VMware [7]. We refer this as receive data combining. Such an enhancement is conceptually similar to TCP's Nagle algorithm [19] in that it attempts to increase efficiency by handling multiple packets together. However, this could result in ACKs being compressed in time.

Interestingly we also see that the ACK stream is smoothed out. The distribution seen at the VM host (Figure 6) shows that the ACK arrivals are aligned along the multiples of one way packet transmission time. But, Figure 7 shows that the interarrival distribution at the VM is spread over the range of interarrival times. The receipt of a data packet requires at least two world switches [7]. 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 6 to 0.26ms and 0.28ms in Figure 7.

In summary, a TCP ACK stream is perturbed in two ways: First, ACKs that arrive for a VM that is waiting to be dispatched are processed together. This batching behavior induces ACK compression on the TCP connection. Secondly, ACKs that are not batched are subject to random delay whose magnitude depends on the system load. This tends to move the ACK interarrival distribution observed at the VM host to the right and to also flatten the distribution modes. This effect can

potentially decrease application throughput although in some situations it might be beneficial to smooth a bursty outbound data stream.

Next, we empirically show that both the effects increase as the number of active virtual machines increase. Figures 7 to 10 show the interarrival distribution of the acknowledgments as seen at vm1 as a function of the number of active VMs. It is clearly evident that the level of ACK-compression and the number of active VMs are highly correlated. Further, we found that the distributions seen at the host are very similar to the baseline assuring us that the ACKs are not compressed in the network but rather within the VM. Figure 10 shows a large mode centered at 0.045ms. It shows that about 18% of ACKs arrive at vm1 with a spatial gap of 0.045ms when 7 VMs are active as against 15% when 4 VMs (Figure 9) are active and 8% when only 1 VM (Figure 7) is active. The increase in per-packet delay is also clearly evident from the figures. The increase in the size of the tail of the interarrival distribution in Figure 10 indicates a significant increase in the delay caused by the virtualization overhead.

A modern OS subject to very high loads is also susceptible to large delays and hence might induce ACK-compression. However, for comparable network loads the delays in a VM system are much larger due to the overhead associated with the world switches involved in the functioning of a VM. Hence, for comparable network loads, a no VM system and a VM system would exhibit very different system dynamics that impact the network differently.

5. Conclusions

We have shown that the virtualization overhead can significantly affect the ACK stream destined for a virtual machine. We have empirically shown that the VM overhead leads to a batching effect and a smoothing effect. These effects are directly proportional to the number of active virtual machines. We have conjectured that this result is caused by 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 were not able to confirm this. It is clear however that these effects become extreme as the number of active VMs grows.

VMs are being widely deployed as low cost Unix Web hosting platforms. There are no provisioning guidelines that have been proposed by either the IT or the research communities. Our research provides a foundation for further work in this area.

References:

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

[2] 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.

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

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

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

[6] Redwoodvirtual.com

[7] 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.

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

[9] 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.

[10] J. Martin, V. Rajasekaran, J. Westall, "Virtual Machine Effects on TCP Perfomance", under review.

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

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

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

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

[15] J. Mogul, "Observing TCP Dynamics in Real Networks", Technical Report, Digital Western Lab, April 1992.

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

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

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

[19] W. R. Stevens, TCP Illustrated, Vol. 1, 1994