Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor

Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim
VMware, Inc.
3145 Porter Dr, Palo Alto, CA 94304
{*yoel,ganesh,bhlim*}*@vmware.com*

Abstract

Virtual machines were developed by IBM in the 1960's to provide concurrent, interactive access to a mainframe computer. Each virtual machine is a replica of the underlying physical machine and users are given the illusion of running directly on the physical machine. Virtual machines also provide benefits like isolation and resource sharing, and the ability to run multiple flavors and configurations of operating systems. VMware™Workstation brings such mainframe-class virtual machine technology to PC-based desktop and workstation computers.

This paper focuses on VMware Workstation's approach to virtualizing I/O devices. PCs have a staggering variety of hardware, and are usually pre-installed with an operating system. Instead of replacing the pre-installed OS, VMware Workstation uses it to host a user-level application (VMApp) component, as well as to schedule a privileged virtual machine monitor (VMM) component. The VMM directly provides high-performance CPU virtualization while the VMApp uses the host OS to virtualize I/O devices and shield the VMM from the variety of devices. A crucial question is whether virtualizing devices via such a hosted architecture can meet the performance required of high throughput, low latency devices.

To this end, this paper studies the virtualization and performance of an Ethernet adapter on VMware Workstation. Results indicate that with optimizations, VMware Workstation's hosted virtualization architecture can match native I/O throughput on standard PCs. Although a straightforward hosted implementation is CPU-limited due to virtualization overhead on a 733 MHz Pentium®III system on a 100 Mb/s Ethernet, a series of optimizations targeted at reducing CPU utilization allows the system to match native network throughput. Further optimizations are discussed both within and outside a hosted architecture.

1 Introduction

The concept of the virtual machine was invented by IBM as a method of time-sharing extremely expensive mainframe hardware [4, 5]. As defined by IBM, a "virtual machine" is a fully protected and isolated copy of the underlying physical machine's hardware. Thus, each virtual machine user is given the illusion of having a dedicated physical machine. Software developers can also write and test programs without fear of crashing the physical machine and affecting the other users.

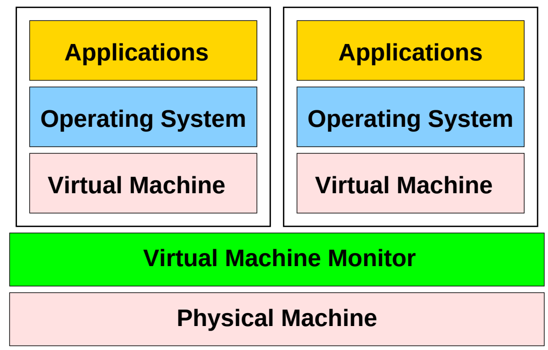

Figure 1: A virtual machine monitor provides a virtual machine abstraction in which standard operating systems and applications may run. Each virtual machine is fully isolated from the rest of the virtual machines.

Figure 1 illustrates the traditional organization of a virtual machine system. A software layer called a *virtual machine monitor (VMM)* takes complete control of the machine hardware and creates virtual machines, each of which behaves like a complete physical machine that can run its own operating system (OS). Contrast this with a normal system where a single operating system is in control of the machine.

To maximize performance, the monitor gets out of the way whenever possible and allows the virtual machine to execute directly on the hardware, albeit in a non-privileged mode. The monitor regains control whenever the virtual machine tries to perform an operation that may affect the correct operation of other virtual machines or of the hardware. The monitor safely emulates the operation before returning control to the virtual machine. This direct execution property allows mainframe-class virtual machines to achieve close to native performance and sets the technology apart from machine emulators that always impose an extra layer of interpretation on the emulated machine.

The result of a complete machine virtualization is the creation of a set of virtual computers that runs on a physical computer. Different operating systems, or separate instances of the same operating system, can run in each virtual machine. The operating systems that run in virtual machines are termed *guest* operating systems. Since virtual machines are isolated from each other, a guest operating system crash does not affect the other virtual machines. Users in different virtual machines cannot affect each other catastrophically.

Most of the benefits of mainframe virtual machines apply to the PC platform, and several new ones have emerged. On mainframes, virtual machines have been used for timesharing, for partitioning machine resources among different OSes and applications, as well as for OS and software development and easing system migration. On a desktop or workstation PC there is a need to run different operating systems – primarily the various flavors of Microsoft® and UNIX™-based operating systems. Virtual machines allow these OSes to be run simultaneously on a single computer.

Intel®-based PCs are also increasingly being used as servers by traditional enterprises and service providers to host applications. Frequently, an entire machine is dedicated to a particular service, application or customer in order to provide fault isolation and performance guarantees. In this arena, virtual machines can be used to host applications, provide better resource utilization, and ease system manageability. Virtual machines can also be easily migrated and replicated across machines to aid in service provisioning. Virtual machines can contain identical virtual hardware, even on hosts with different native hardware, making virtual machines freely portable between different physical machines.

1.1 Virtualizing the PC platform

Several technical and pragmatic hurdles must be overcome when virtualizing the PC platform. The traditional mainframe approach runs virtual machines in a less privileged mode in order to allow the VMM to regain control on privileged instructions, and relies on the VMM to virtualize and interface directly to the I/O devices. Also, the VMM is in complete control of the entire machine. This approach doesn't apply as easily to PCs for the following reasons.

Non-virtualizable processor – The Intel IA-32 processor architecture [10] is not naturally virtualizable. Popek and Goldberg [11] showed that an architecture can support virtual machines only if all instructions that can inspect or modify privileged machine state will trap when executed from any but the most privileged mode. Because the IA-32 processor does not meet this condition, it is not possible to virtualize the processor by simply executing all virtual machine instructions in a less privileged mode.

PC hardware diversity – There is a large diversity of devices that may be found in PCs. This is a result of the PC's "open" architecture. In a traditional implementation, the virtual machine monitor would have to manage these devices. This would require a large programming effort to provide device drivers in the VMM for all supported PC devices.

Pre-existing PC software – Unlike mainframes that are configured and managed by experienced system administrators, desktop and workstation PC's are often pre-installed with a standard OS and set up and managed by the end-user. In this environment, it is extremely important to allow a user to adopt virtual machine technology without losing the ability to continue using his existing OS and applications. It would be unacceptable to completely replace an existing OS with a virtual machine monitor.

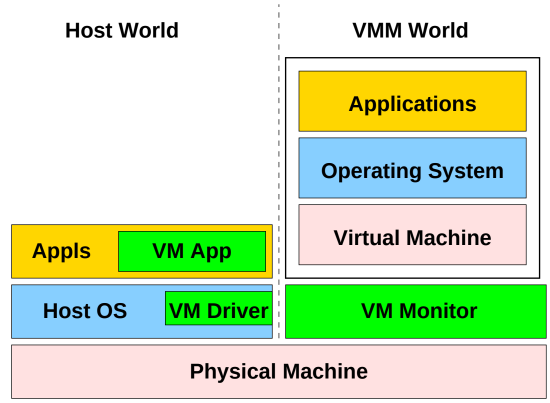

Figure 2: VMware's hosted virtual machine model splits the virtualization software between a virtual machine monitor that virtualizes the CPU, an application the uses a host operating system for device support, and an operating system driver for transitioning between them.

VMware Workstation has a *hosted* architecture that allows it to co-exist with a pre-existing *host* operating system, and rely upon that operating system for device support. Figure 2 illustrates the components of this hosted architecture. This architecture allows VMware to cope with the diversity of PC hardware and to be compatible with pre-existing PC software. Currently, Windows NT®, Windows®2000 and Linux can serve as hosts. This paper focuses on the performance aspects of relying on a host OS for accessing I/O devices.

The rest of this paper is organized as follows. Section 2 describes VMware Workstation's hosted architecture, its benefits and costs, and looks at the specific example of a virtual Ethernet network interface card (NIC). Section 3 demonstrates the performance of NIC virtualization with VMware Workstation 2.0, breaks down the overheads for a few different workloads, and measures improvements achieved by optimizations the data suggested. Section 4 presents several approaches for improving I/O performance beyond the optimizations described in Section 3, some of which go beyond the capabilities of a hosted architecture. Section 5 describes related work in the area of supporting multi-platform computing on a single machine. Finally, Section 6 summarizes the observed properties of the hosted architecture and draws some conclusions about this approach to I/O virtualization.

2 A Hosted Virtual Machine Architecture

VMware Workstation virtualizes I/O devices using a novel design called the *Hosted Virtual Machine Architecture*. The primary feature of this design is that it takes advantage of a pre-existing operating system for I/O device support and still achieves near native performance for CPU-intensive workloads. Figure 2 illustrates the structure of a virtual machine in the hosted architecture.

VMware Workstation installs like a normal application on an operating system, known as the host operating system. When run, the application portion (**VMApp**) uses a driver loaded into the host operating system (**VMDriver**) to establish the privileged virtual machine monitor component (**VMM**) that runs directly on the hardware. From then on, a given physical processor is executing either the host world or the VMM world, with the VMDriver facilitating the transfer of control between the two worlds. A world switch between the VMM and the host worlds involves saving and restoring *all* user and system visible state on the CPU, and is thus more heavyweight than a normal process switch.

In this architecture, the CPU virtualization is handled by the VMM. A guest application or operating system performing pure computation runs just like a traditional mainframe-style virtual machine system. However, whenever the guest performs an I/O operation, the VMM will intercept it and switch to the host world rather than accessing the native hardware directly. Once in the host world, the VMApp will perform the I/O on behalf of the virtual machine through appropriate system calls. For example, an attempt by the guest to fetch sectors from its disk will become a `read()` issued to the host for the corresponding data. The VMM also yields control to the host OS upon receiving a hardware interrupt. The hardware interrupt is reasserted in the host world so that the host OS will process the interrupt as if it came directly from hardware.

The hosted architecture is a powerful way for a PC-based virtual machine monitor to cope with the vast array of available hardware. One of the primary purposes of an operating system is to present applications with an abstraction of the hardware that allows hardware-independent code to access the underlying devices. For example, a program to play audio CD-ROMs will work on both IDE and SCSI CD-ROM drives because operating systems provide an abstract CD-ROM interface. VMware Workstation takes advantage of this generality to run on whole classes of hardware without itself needing special device drivers for each possible device.

The most significant trade-off of a hosted architecture is in potential I/O performance degradation. Because I/O emulation is done in the host world, a virtual machine executing an I/O intensive workload can accrue extra CPU time switching between the VMM and host worlds, as well as significant time in the host world performing I/O to the native hardware. This increases the CPU overhead associated with any I/O operation.

Another trade-off of the hosted architecture is that the host OS is in full control of machine resources. Even though the VMM has full system and hardware privileges, it behaves cooperatively and allows the host OS to schedule it. The host OS can also page out the memory allocated to a particular virtual machine except for a small set of pages that the VMM has pinned on behalf of the virtual machine. This allows VMware Workstation to be treated by the host OS like a regular application, but occasionally at the expense of performance if the host OS makes poor resource scheduling choices for the virtual machine.

2.1 Virtualizing I/O Devices

Every VMware virtual machine is configured from the same set of potential virtual devices. Supported are standard PC devices such as a PS/2 keyboard, PS/2 mouse, floppy drive, IDE controllers with ATA disks and ATAPI CD-ROMs, a Soundblaster 16 sound card, and serial and parallel ports. Each virtual machine can also populate its virtual PCI slots with virtual BusLogic SCSI controllers, AMD PCNet™ Ethernet adapters, and an SVGA video controller for a special VMware virtual display card. Note that since the hardware besides the SVGA controller is made up of standard PC devices, existing guest operating system device drivers can communicate with it without modification.

In order to virtualize an I/O device, the VMM must be able to intercept all I/O operations issued by the guest operating system. On a PC, those accesses are generally done via special privileged IA-32 IN and OUT instructions. These are trapped by the VMM and emulated either in the VMM or the VMApp by software that understands the semantics of the specific I/O port accessed. Any accesses that interact with the physical I/O hardware must be handled in the VMApp, but the VMM can potentially handle accesses that do not interact with the hardware, *e.g.*,

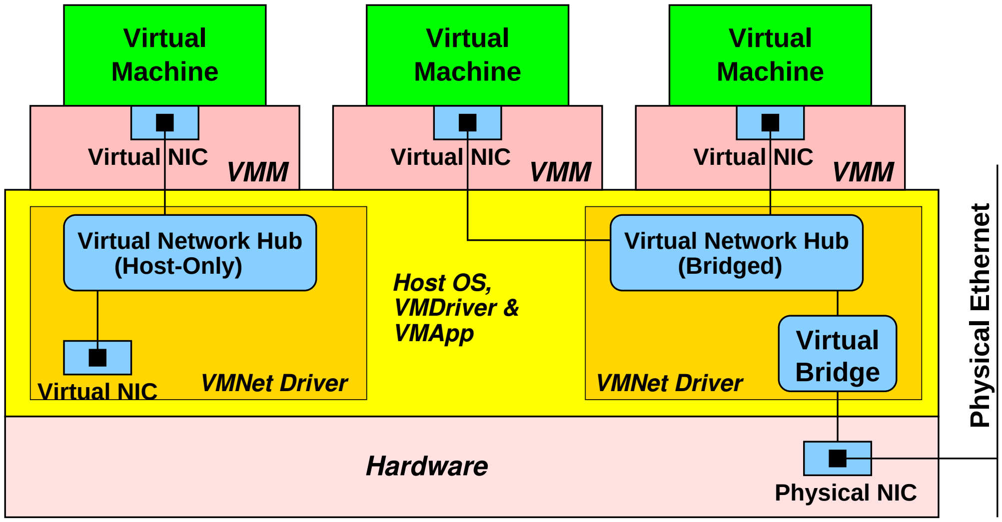

Figure 3: VMware's network subsystem provides virtual Ethernet adapters, hubs and bridges. A hub can be either be bridged to a physical Ethernet adapter, or connected to a virtual network interface in the host OS. The virtual bridge and hub are implemented via a VMNet driver that is loaded into the host OS.

status ports or ports that merely latch data that will be used later. Restricting virtual devices to only a subset of available PC hardware greatly reduces the number of I/O ports that must be handled and the breadth of possibilities that handlers need to understand.

Virtualizing I/O devices with the hosted architecture can incur overhead from world switches between the VMM and the host, and even from the expense of handling the privileged instructions used to communicate with the hardware. However, these overheads matter only for devices with either high sustained throughput or low latency. The keyboard, for example, is perfectly suited to hosted virtualization.

2.2 Virtualizing a Network Card

An excellent example of a device that requires both high sustained throughput and low latency is a network interface card (NIC). Therefore, to understand how hosted device virtualization works and its performance implications, the following sections focus on the specific example of emulating a NIC in VMware Workstation. Figure 3 illustrates the components of the system. The virtual NIC appears to the guest as a full-fledged PCI Ethernet controller, complete with its own MAC address. The NIC emulation can be connected to the host in two ways– it can be bridged to the same physical network as a physical NIC or it can be connected to a virtual network created on the host. In both cases, the connection is implemented by a VMware *VMNet* driver that is loaded in the host operating system.

A virtual NIC that is bridged to a physical NIC is a true Ethernet bridge in the strictest sense. Its packets are sent on the wire with its own unique MAC address. The VMNet driver runs the bridged physical NIC in promiscuous mode so that replies to that MAC address are picked up. The virtual NIC appears on the local Ethernet segment indistinguishably from any real machine. As a result, a virtual machine with a bridged virtual NIC can fully participate in accessing and providing network services.

A virtual NIC that is connected to a virtual network does not require an Ethernet interface on the host. Unlike the bridged case, the virtual network is completely private within the host and any participating virtual machines. If desired, the host OS can perform routing or IP masquerading to connect a virtual network to any type of external network, even to a non-Ethernet network. This paper will focus only on virtual NICs bridged to a physical NIC.

A virtual NIC itself is implemented via a combination of code in the VMM and the VMApp. The VMM exports a number of virtual I/O ports and a virtual IRQ that represent the virtual network adapter in the virtual machine. Reads and writes to these I/O ports, as well as virtual DMA transfers between the adapter and the virtual machine's memory are semantically equivalent to those of a real network adapter. In VMware Workstation, the virtual NIC models an AMD Lance Am79C970A [1] controller, except that it is not limited to any specific network speed.

2.3 Sending and Receiving via a Virtualized NIC

Figure 4 depicts the components involved when sending and receiving packets via the hosted virtual NIC emulation described above. The guest operating system runs the device driver for a Lance controller. The driver initiates packet transmissions by reading and writing a sequence of virtual I/O ports, each of which switches back

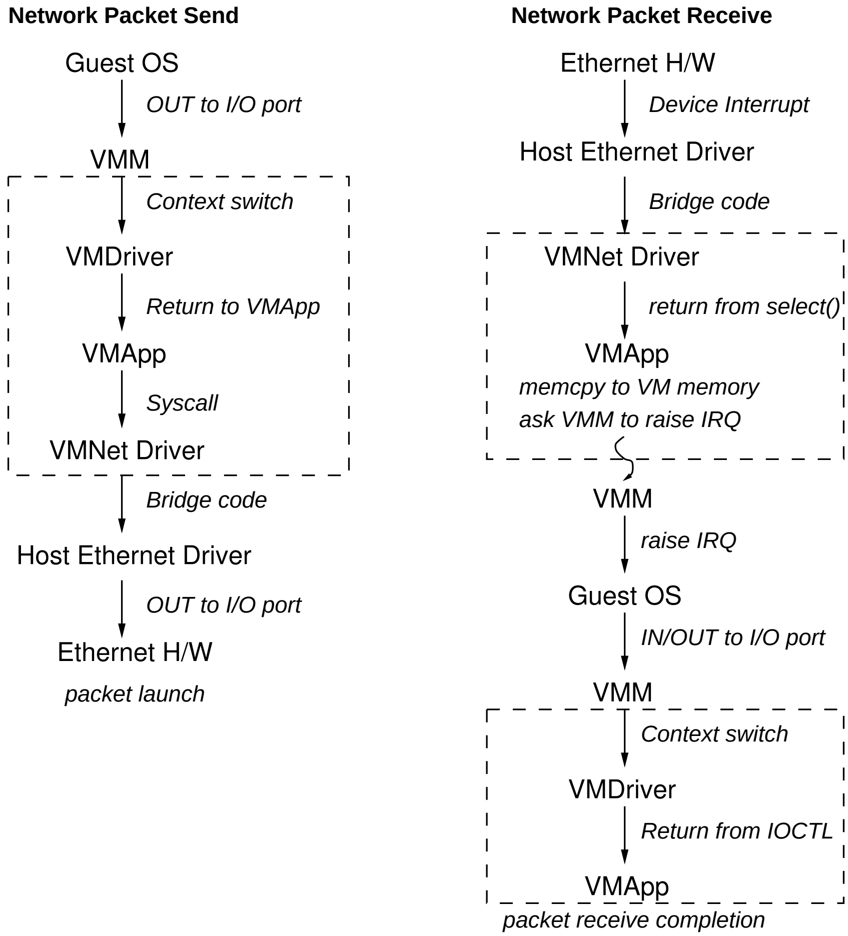

Figure 4: Components involved in a virtual machine network packet send and receive. Boxed components delineate components that are due to the hosted nature of the network device virtualization.

to the VMApp to emulates the Lance port accesses. On the final OUT instruction of the sequence, the Lance emulation does a normal `write()` to the VMNet driver, which passes the packet onto the network via a host NIC and then the VMApp switches back to the VMM, which raises a virtual IRQ to notify the guest device driver the packet was sent.

Packet receives occur in reverse. The bridged host NIC delivers the packet to the VMNet. The VMApp periodically runs `select()` on its connection to the VMNet and `read()`s the packet and requests that the VMM raise a virtual IRQ when it discovers any incoming packets. The VMM posts the virtual IRQ and the guest's Lance driver issues a sequence of I/O accesses to acknowledge the receipt to the hardware.

The boxed regions of the figure indicate extra work introduced by virtualizing the port accesses that actually send and receive packets. There is additional work in handling the intermediate I/O accesses and the privileged instructions associated with handling a virtual IRQ. Of the intermediate accesses, the ones to the virtual Lance's address register are handled completely within the VMM and all accesses to the data register switch back to handling code in the VMApp.

This extra overhead consumes CPU cycles and increases the load on the CPU. The next section studies the effect of this extra overhead on I/O performance as well as CPU utilization. It breaks down the overheads along the boxed paths and describes overall time usage in the VMM and VMApp during the course of network activity.

3 Virtual Machine Networking Performance

A hosted virtualization strategy for I/O devices offers excellent flexibility and portability but at a potential tradeoff in performance for high throughput devices. Due to its nature, the hosted architecture incurs the following overheads: i) a world switch from the VMM to the host is required whenever the virtual machine needs to access real hardware, ii) I/O interrupt handling potentially involves the VMM, host OS, and guest OS interrupt handlers, iii) a packet transmission by the guest OS involves two device

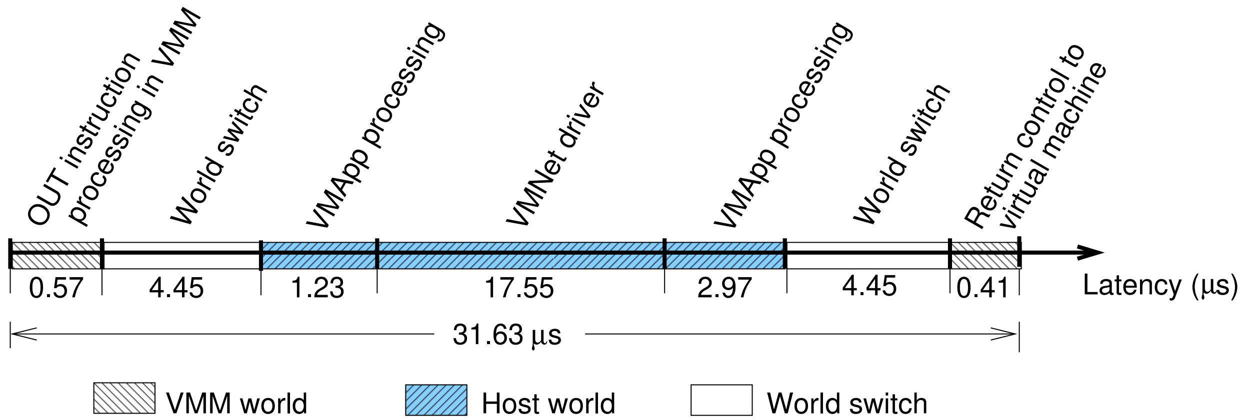

Figure 5: Microseconds spent along the path in the Host and VMM worlds processing an OUT instruction issued by the guest OS to a virtual AMD Lance NIC that initiates a physical network packet transmission on a 733 MHz CPU machine.

drivers - one in the guest and one on the host, and iv) there is an extra copy from the guest OS's physical memory to the host OS's kernel buffers on a packet transmit. Since these overheads consume CPU cycles, a system that is natively capable of saturating a high performance Ethernet link might instead become CPU bound when run within a virtual machine.

This section analyzes the overheads of sustained TCP transmits from a virtual machine. Experimental results of sustained TCP receives yield similar results and conclusions, and are not presented here due to space constraints. An analysis of these workloads exposes the major sources of virtualization overhead on a hosted architecture. Frequent switches between the host and VMM worlds is the most significant overhead. A set of optimizations targeted at these overheads improves the virtual networking subsystem substantially. The experimental results show that a set of three optimizations doubles sustained TCP transmit throughput on a slower machine that is CPU bound, and reduces CPU utilization significantly on a faster machine that is I/O bound.

3.1 Experimental Setup

The experiments were performed on two Intel-based PCs that are physically connected to each other via Intel EtherExpress 100 Mb/s Ethernet NICs and a direct, crossover cable:

PC-350 – a 350 MHz Pentium II system with 128 MBytes of RAM running a Linux 2.2.13 kernel

PC-733 – a 733 MHz Pentium III system with 256 MBytes of RAM running a Linux 2.2.17 kernel.

A virtual machine is configured with a virtual AMD Lance NIC bridged to the native Intel EtherExpress NIC. The virtual machine runs a standard RedHat 6.2 Linux guest OS plus the 2.2.17-14 kernel update and uses the standard Linux `pcnet32` driver to communicate over the virtual network. This virtual machine is hosted in two configurations on VMware Workstation 2.0:

VM/PC-350 – the virtual machine with 64 MBytes of RAM hosted by **PC-350**.

VM/PC-733 – the virtual machine with 128 MBytes of RAM hosted by **PC-733**.

The throughput experiments use a simple program called `nettest` that was developed internally for benchmarking network performance. The program opens a TCP connection between two IP addresses and copies a user specified amount of data with individual `send()`s and `recv()`s of a user specified size. The data transferred is merely repeated copies of the same in-memory buffer (to avoid paging and disk overhead) and is discarded on the receive side as soon as it arrives. The program measures the entire transfer and reports the average throughput in Mb/s.

3.2 Packet Transmit Overheads

The first series of experiments investigates the behavior of sustained virtual machine TCP transmits from **VM/PC-733** to **PC-350**. We configure `nettest` to send 100 megabytes using 4096-byte `read()`s and `write()`s. With VMware Workstation 2.0, we find that the workload is CPU bound with an average throughput over 30 consecutive runs of 64 Mb/s.

The workload is then instrumented to determine where the CPU time is spent. The first instrumentation gauges the time spent transmitting a packet by reading the the Pentium Processor's *Time Stamp Counter* (TSC) register [10] at key points during the virtualization of the OUT instruction that

Total Time		
Category	Percent Time	Average Time
VMM Time	77.3%	N/A
Transmitting via the VMNet	8.7%	13.8 μs
Emulating the Lance status register	4.0%	3.1 μs
Handling host IRQs (device interrupts)	3.4%	N/A
Emulating the Lance transmit path	3.3%	5.2 μs
Receiving via the VMNet	0.8%	1.8 μs
VMM Time		
Category	Percent Time	Average Time
IN/OUTs requiring switching to the VMApp	26.8%	7.45 μs
Instructions not requiring virtualization	22.0%	N/A
General instructions requiring virtualization	11.6%	N/A
IN/OUTs handled in the VMM	8.3%	1.36 μs
IN/OUTs to the Lance Address Port	8.1%	0.74 μs
Transitioning to/from virtualization code	4.8%	N/A
Virtualizing the IRET instruction	4.8%	3.93 μs
Delivering virtual IRQs (device interrupts)	4.6%	N/A

Table 1: Distribution of CPU time during network transmission. The largest overheads are I/O space accesses requiring a world switch to the VMApp and the time spent handling them once in the VMApp.

triggers a packet transmission. The TSC allows a measurement of the total cycle count of the path, plus internal breakdowns of interesting subsegments.

Figure 5 presents the latency involved along the instrumented network transmit path on **PC-733**. It takes a total of 0.57+4.45+1.23+17.55 = 23.8 μs from the start of the OUT instruction until the return from the VMNet system call that puts the packet on the wire. End-to-end, it takes 31.63 μs from the start of the OUT instruction that triggers a packet transmission until control is returned to the virtual machine and the next guest OS instruction is executed.

Of those 31.63 μs, 30.65 μs is spent in world switches and in the host world. Assuming the 17.55 μs of VMNet driver time in the host world is dominated by the unavoidable cost of actually transmitting the packet, we find that hosted virtualization architecture imposes 30.65 - 17.55 = 13.10 μs of overhead that would not be present if the VMM talked directly to the host NIC.

This overhead alone does not explain why the workload is CPU bound. At 31.63 μs per 1520-byte packet, it only takes roughly 0.26 seconds to transmit 100 megabits. Each packet transmission actually involves a series of 11 other IN/OUT instructions issued by the guest Ethernet driver as well as interrupt processing and virtualization overheads.

To investigate these other overheads, the next set of experiments uses time-based sampling to profile the distribution of time spent in the VMM and VMApp over the entire workload. The samples measure the percentage of time spent in code sections and the number of samples that hit a section (when available). This gives a more comprehensive picture of the overheads present in transmitting packets and reveals some unnecessarily expensive paths. Table 1 summarizes the highest categories.

The profile shows that more than a quarter of the time in the VMM is spent preparing to call the VMApp because of an I/O instruction, recording the result and then returning to the virtual machine. Additionally, each of those transitions also cost a world switch from the VMM to the host and back, which was calculated at around 8.90 μs on **PC-733** above (the switch time is part of the 77.3% running the VMM, but not part of any of the VMM Time numbers). Given that an I/O instruction on native hardware completes in a matter of tens of cycles, this is easily two orders of magnitude slower.

The other significant source of overhead is spread through the categories in Table 1: IRQ processing. The virtual AMD Lance NIC as well as the physical Intel EtherExpress NIC raises an IRQ (device interrupt) on every packet sent and received. Thus, the interrupt rate on the machine is very high for network-intensive workloads. On a hosted architecture, each IRQ that arrives while executing in the VMM world runs the VMM's interrupt handler then switches to the host world. The host world runs the host OS's interrupt handler for that IRQ, and passes control to the VMApp to process any resulting actions. If the IRQ pertains to the guest (*e.g.*, the IRQ indicates that a packet was received that is destined for the guest), the VMApp will then need to deliver a virtual IRQ to the guest OS. This involves switching back to the VMM world, delivering an IRQ to the virtual machine, and running the guest OS's interrupt handler.

This magnifies the cost of an IRQ since VMM and host interrupt handlers as well as guest interrupt handlers are run. Additionally, virtual interrupt handling routines execute privileged instructions that are expensive to virtualize. In Table 1, most of the IN/OUTs handled in the VMM are accesses to the virtual interrupt controller and the majority of the IRET instructions are the guest interrupt handler finishing. Note also that the cost of servicing an interrupt taken in the VMM world is much higher than servicing an interrupt taken in the host world due to the VMM interrupt handler and a world switch back to the host.

Yet another overhead in the hosted architecture which is not apparent from the raw profile is the inability of the VMApp and VMM to distinguish between a hardware interrupt which produces an event for the virtual machine (*e.g.*, a packet to be delivered to the guest was received) from one that is unrelated to the virtual machine. Only the host OS and its drivers determine that. This leads to a balancing act: The VMApp can do nothing when the VMM returns to the VMApp on an IRQ, or it call `select()` in the VMApp. Calling `select()` too frequently is wasteful, whereas calling `select()` too infrequently may cause harmful delays in handling network I/O events.

3.3 Reducing Network Virtualization Overheads

Guided by the results from the previous subsection that show world switch overheads as having the biggest impact, we implemented a set of optimizations aimed at reducing the number of world switches dramatically without departing from the hosted I/O architecture.

Handling I/O ports in the VMM Recall that the only virtual I/O accesses that require a world switch to the host are the ones that require a physical I/O device access. The vast majority of the I/O instructions are accesses to the Lance data port and only a third of them trigger packet transmissions. The remaining accesses merely modify the state of the virtual Lance data port, which can be easily done directly in the VMM without a world switch. Thus, an emulation of an OUT instruction that does not require real I/O can now be achieved in less than a tenth of the time it takes in VMware Workstation 2.0.

We also further reduce the cost of processing I/O accesses to the Lance address port by taking advantage of the property that the Lance address register has memory semantics, *i.e.*, reads and writes have no side effects and only latch the last value written. Thus, even though the instructions to access it are privileged instructions, the VMM can treat them as simple MOV instructions that happen to store to a special location. This allows the VMM to strip away several layers of virtualization and reduce the handling of the accesses to a handful of instructions.

Send combining The second optimization further reduces world switches by taking advantage of the fact that I/O intensive workloads have a high interrupt rate and the VMM must switch to the host whenever it takes a host IRQ. In VMware Workstation 2.0, each packet sent on the Lance adapter causes a world switch to the host to send the packet over the bridged network. Since part of the Lance data port emulation is now performed in the VMM, the VMM can delay the actual transmission until the next interrupt-induced switch to the host world.

Specifically, send combining work as follows: the VMM detects whether the system is experiencing a high world switch rate. If the rate (as recalculated periodically with an exponentially decaying counter) is high enough when the guest transmits a packet, the VMM queues it in a ring buffer and resumes the virtual machine. The next time a real interrupt occurs and control returns to the VMApp it transmits any pending packets in the ring buffer. This effectively allows a packet transmission world switch to be combined with an interrupt-induced one.

Queueing the packets can be done without copying by leaving them in the virtual Lance controller's transmit ring buffer. If too many packets are delayed (currently defaulting to 3), the VMM will force a world switch to transmit the packets in order to insure that the native NIC is kept busy. In addition, there is a guaranteed world switch on the next IRQ from the host system timer so no packet will ever be delayed more than one tick (at which point the VMApp will discontinue send combining if necessary). This optimization works well on I/O intensive workloads because interrupt rates are high enough that world switches are saved while I/O utilization is sustained.

Send combining also benefits both guest and host IRQs. Since the guest continues executing as soon as the packets are queued, there is a high probability that the guest will transmit multiple packets before the next mandatory world switch. This allows the VMApp to process multiple transmit packets on a single world switch and deliver only a single virtual IRQ for the batch. As noted earlier, virtual IRQ delivery and the associated privileged virtualization are expensive operations. Furthermore, transmitting multiple packets at once increases the probability that native send-complete interrupts are taken while executing in the host world and hardware interrupts taken in the host world are serviced faster than those taken in the VMM world.

IRQ notification The third optimization is targeted at reducing host system calls for receiving notification of packet sends and receives. The VMApp establishes a piece of shared memory with the VMNet driver at initialization and the driver sets a bitvector whenever packets are available. Then, on every NIC IRQ, instead of an expensive `select()` on all of the devices, the VMApp checks the shared memory, receives any pending packets, and immediately returns to the VMM.

Total Time		
Category	Percent Time	Average Time
VMM Time	71.5%	N/A
Transmitting via the VMNet	17.9%	22.7 μs
Receiving via the VMNet	2.5%	22.7 μs
Emulating the Lance transmit path	1.8%	3.0 μs
VMM Time		
Category	Percent Time	Average Time
Guest idle*	30.4%	N/A
Instructions not requiring virtualization	22.2%	N/A
Guest context switches	11.5%	N/A
Host IRQ processing while guest idle*	10.7%	N/A
Virtualizing privileged instructions	7.9%	N/A
IN/OUTs to the PIC (Interrupt Controller)	2.5%	0.78 μs
Virtualization overheads of guest IRQs	2.5%	N/A
IN/OUTs to the Lance status register	2.3%	0.91 μs
Transitioning to/from virtualization code	1.5%	N/A
IN/OUTs to the Lance that world switch	0.5%	N/A

Table 2: Distribution of CPU time during network transmission, with VMM optimizations. Many of the VMM time entries now represent a collection of individual instructions, which renders the Average Time not applicable. *Categories marked with '*' are partly derived from direct measurements presented in Section 3.5 for reasons described below.

In summary, the three major optimizations applied are as follows: Lance related I/O port accesses from the virtual machine are handled in the VMM whenever possible. During periods of heavy network activity, packet transmissions are merged and sent during IRQ-triggered world switches. This reduces the number of world switches, the number of virtual IRQs, and the number of host IRQs taken while executing in the VMM world. Finally, the VMNet driver is augmented with shared memory that allows the VMApp to avoid calling `select()` in some circumstances.

Figure 6 shows that these optimizations reduce CPU overhead enough to allow **VM/PC-733** to saturate a 100 Mbit Ethernet link, and the throughput for **VM/PC-350** more than doubles. Table 2 lists the CPU overhead breakdown from the time-based sampling measurements on **VM/PC-733** with the optimizations in place. Overall, the profile shows that the majority of the I/O related overhead is gone from the VMM and that there is now time when the guest OS idles. Additionally, guest context switch virtualization overheads now become significant as the guest switches between `nettest` and its idle task.

The "Guest idle" and "Host IRQ processing while guest idle" categories in Table 2 are derived with input from direct measurements presented in Section 3.5. A sample-based measurement of idle time indicates that 41.1% of VMM time is spent idling the guest and taking host IRQs while idling. However, discriminating the host IRQ processing time and guest idle time via time-based sampling alone is hard because of synchronized timer ticks and the heavy interrupt rate produced by the workload. We use direct measurements that show that 21.7% of *total* time is spent in the guest idle loop to arrive at the idle time breakdown in Table 2.

The most effective optimization is handling IN and OUT accesses to Lance I/O ports directly in the VMM whenever possible. This eliminates world switches on Lance port access that do not require real I/O. Additionally, Table 1 indicates that accessing the Lance address register consumes around 8% of the VMM's time and taking advantage of the register's memory semantics has completely eliminated that overhead from the profile as shown in Table 2.

An interesting observation is that the time to transmit a packet via the VMNet does not change noticeably – all of the gains are along other paths. Instrumenting the optimized version in appropriate locations shows that the average cycle count on the path to transmit a packet onto the physical NIC is within 100 cycles of the totals from Figure 5. However, this is contrary to the times in Table 2 for sending via the VMNet driver. This disagreement stems from transmitting more than one packet at a time. While simply sending and timing individual packets, the baseline and optimized transmits look very similar, but with send combining active, up to 3 packets are sent back to back. This increases the chance of taking a host transmit IRQ from a prior transmit while in the VMNet driver. Since Table 2 reports the time from the start to finish of the call into the VMNet driver, it also includes the time the host kernel spends handling IRQs.

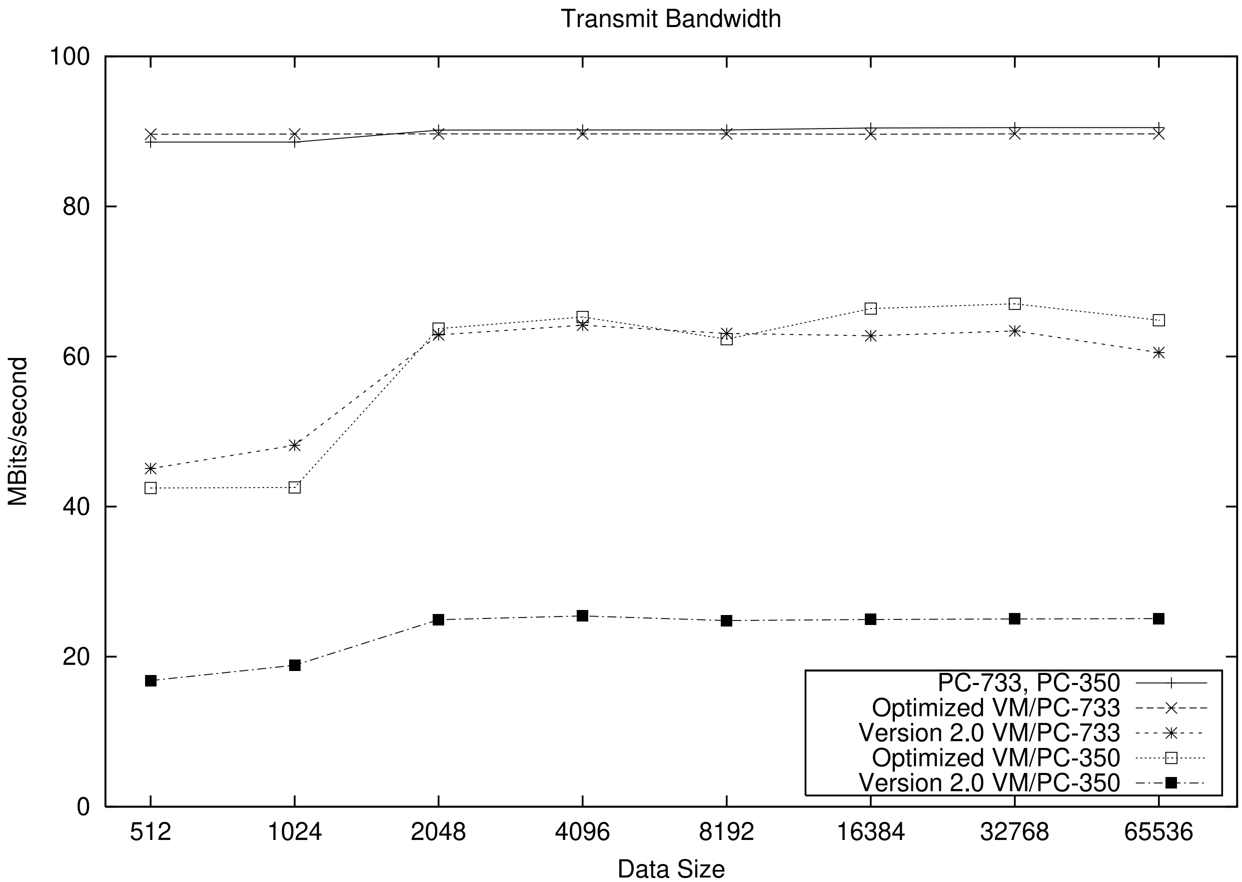

Figure 6: Throughput vs. Data Size when transmitting. The native and **VM/PC-733** lines are transmitting to **PC-350** and the **VM/PC-350** lines are transmitting to **PC-733**. The optimized **VM/PC-733** was able to achieve native speeds, but non-I/O virtualization overheads limited **VM/PC-350**'s achievable I/O throughput.

3.4 Throughput vs. Data Size: Transmit

The next series of experiments investigates the effect of the per-`write()` data size on the overall throughput. The data was gathered with the same `nettest` program and 100 MByte copies, but the amount of data copied per `write()` was varied from 512 to 65536 bytes by powers of two. 30 runs were taken and averaged at each data size with both the optimized and 2.0 versions. Figure 6 shows the sustained transmit throughput from the various machine configurations and optimization levels.

As expected, the native machines (both **PC-733** and **PC-350** achieve identical throughput) saturate the 100 Mbit link. **VM/PC-733** becomes CPU bound well before saturating the network link. With the optimizations however, **VM/PC-733** matches native throughput. Although **VM/PC-350** remains CPU bound with the optimizations, its sustained throughput doubles and matches the performance of the unoptimized **VM/PC-733**. The two **VM/PC-350** curves are consistent in shape with their **PC-733** counterparts.

3.5 CPU Utilization

Figure 6 shows that **VM/PC-733** is able to saturate a 100 Mbit link without becoming CPU bound, but **VM/PC-350** is CPU bound, even with optimizations. Natively, **PC-733** and **PC-350** easily saturate a 100 Mbit link. The final experiments set out to gather information about how utilized the CPU is in the different configurations.

We instrumented the system to obtain a precise measurement of idle time. Normally, when a guest issues a halt (HLT) instruction, VMware Workstation switches back to the VMApp which then blocks on a `select()` on all devices. Instead, we enabled an option whereby a guest HLT instruction spins and halts the CPU in the VMM rather than yielding control back to the host OS. Using the TSC register, we measure idle time starting from when the guest issues a HLT instruction to when the next hardware interrupt occurs. This idle time represents CPU cycles that is available to the guest OS for running other computation. Note that not all of this idle time would be available for other host OS computation, as there are a couple of world

Idle Time While Running `nettest`	
PC-733	86%
Optimized **VM/PC-733**	21.7%
Optimized **VM/PC-733** without IRQ notification	17.9%
Optimized **VM/PC-733** without send combining and IRQ notification	2.0%
Version 2.0 **VM/PC-733**	0%

Table 3: Percentage of total time spent idle for various configurations transmitting data on **PC-733**.

switches and some system call overhead (e.g., the `select()` system call) if we switched back to the VMApp on a guest HLT instruction.

For the native idle times, the standard profiler built into Linux kernels was augmented to account for time spent executing user code and in the kernel idle loop, and then the percentage of total ticks spent in the idle loop was taken.

The idle times in Table 3 show that in **VM/PC-733**, with a transmit size of 4KB, the guest has transitioned from being CPU bound at 64 Mb/s to being I/O bound with 21.7% idle time. In comparison, **PC-733** has 86% idle time. At this point, nearly all of the remaining overheads are either part of CPU virtualization or part of the nature of the hosted architecture. The next section discusses further optimizations both within and outside the scope of a hosted architecture.

4 Performance Enhancements

The previous section showed that targeted optimizations can reduce the CPU overhead due to virtualization to the point where performance becomes I/O bound. This section describes strategies for further improving I/O performance and decreasing CPU utilization. The major areas for optimization include i) reducing CPU and interrupt controller virtualization overheads, ii) modifying the guest OS and/or its drivers, iii) modifying the host OS, and iv) accessing the native hardware directly from the virtual machine monitor. The last two techniques are departures from a pure hosted virtual machine architecture. Recall that the hosted architecture is designed with the requirement that existing host operating systems continue to run as usual, and that the virtualization software uses the host OS's API to access hardware devices.

4.1 Reducing CPU Virtualization Overhead

The optimized profile of Table 2 still shows significant overhead for "core CPU virtualization" overheads such as delivering virtual IRQs to a guest operating system, handling IRET instructions, and the MMU overheads associated with context switches. However, a discussion of any of these topics in enough detail to make concrete suggestions requires an understanding of VMware Workstation's core virtualization technology, and is beyond the scope of this paper.

The profile does however suggest one easy optimization to reduce virtualization overhead. Guest OS accesses to the virtual PIC (interrupt controller) accounts for 2.5% of VMM time. A network card saturating a 100 megabit link is transmitting around 8000 packets per second and, in the case of TCP, receiving a steady flow of incoming ACK packets as well. This causes the virtual machine to receive a high rate of virtual IRQs. For each IRQ, the Linux guest IRQ handler issues five accesses to the virtual PIC. Since the virtual PIC is independent of the real PIC, it is handled in the VMM without requiring world switches. We can further optimize these accesses. One of the five accesses has memory semantics and can be inlined as a MOV instruction (just like the Lance address register). The other four accesses cause the current virtual PIC implementation to completely recalculate its internal state in a very general way – this can be specialized to reduce the overhead of those accesses.

4.2 Modifying the Guest OS

It is possible to modify the guest OS to avoid using instructions that virtualize inefficiently. Going a step further, it is also possible to provide a safe call into the VMM from the guest OS to provide some semantic knowledge about the guest to the VMM, or to perform some operations on its behalf. This technique comes at the price of compatibility with off-the-shelf guest OSes.

An optimization we tried in this category is to alter the Linux kernel to avoid page table switches when switching to the idle task. An idle guest spends a significant amount of time context switching to and from its idle task. A guest context switch operation uses a number of privileged instructions and changes guest page tables. This requires VMM intervention to implement the guest context switch safely. In the experiments above, as optimizations are added to reduce CPU utilization, the virtual machine execution profiles show an increasing fraction of CPU overhead due to virtualizing guest context switches. The VMM in **VM/PC-733** spends 8.5% of its time virtualizing page tables switches.

Linux's 2.2 kernels run the idle task as a kernel thread with the kernel's page table. The kernel's page table is a subset of every user application's page table. This implies that it is not necessary to switch page tables when switching to the idle thread. Further, if the idle thread runs with the page table of the last user process to run and the idle thread ends up yielding back to the same process, another page table switch can be avoided. This optimization relies on trusting the idle thread not to corrupt user memory, a reasonable requirement since the idle thread runs at a trusted kernel-level.

We prototyped the optimization of running the idle task with the prior user application's page table by modifying the Linux kernel's context switch function. This modification halves the MMU derived virtualization overhead, and almost all of the saved CPU cycles become CPU idle time. Besides reducing virtualization overhead, such an optimization may also benefit software-based IA-32 CPU implementations where the overhead of emulating the instructions involved in a context switch is significant.

4.3 Optimizing the Guest Driver Protocol

A hosted architecture allows the NIC emulation code to communicate to the host via an abstracted interface that is independent of the host's native hardware. It is possible to design a similarly abstracted imaginary Ethernet controller whose interface is an idealization designed explicitly to virtualize well. For example, the Linux `pcnet32` driver issues 12 I/O instructions and takes one IRQ for every single packet transmitted. An idealized virtual NIC could use only a single OUT to indicate a packet is ready to send and completely skip the transmit IRQ, or to only get an IRQ when space becomes available in the array of outgoing packets. The idealized device can also arrange its transmit and receive buffers very simply in memory rather than with the elaborate flexibility, but complexity of the Lance controller's buffers. In fact, VMware's server products support a `vmxnet` network adapter that implements such an ideal interface.

The major drawback of creating an idealized virtual NIC is the need for custom device drivers for every guest OS. Since the AMD Lance is a well supported NIC, most operating systems already include drivers for it. These existing drivers work *unmodified* in the guest OSes. Any idealized NIC would need to have an array of its own drivers written, distributed, and maintained. Thus, while an idealized driver is a potential accelerating option, it is likely feasible only for critical situations on a select group of guest operating systems, such as in a server environment.

4.4 Modifying the Host OS

Just as expensive virtualization overheads can sometimes be removed by modifying the guest rather than by modifying the VMM, some bumps in the hosted architecture's handling of networking are best smoothed by modifying the host. One promising change is to expand the ways in which the Linux networking stack allocates and handles `sk_buffs`. Each time the VMApp sends a packet via the VMNet driver, the driver allocates an `sk_buff` and copies the data from the VMApp into the `sk_buff`. The Linux kernel profiler shows that a very significant portion of the time spent in the host kernel while running the network transmit workload is due to copying data from the VMApp into an `sk_buff`.

In Linux, `sk_buff` creation uses `kmalloc()` to allocate the data area. If a driver could specify its own data region, then it would be possible to transmit packets via the VMNet driver without the copy. The driver would need to be responsible for making sure that its allocated `sk_buffs` are neither leaked nor freed too early. However, for the VMnet driver, the backing for the `sk_buff` data area would come from the memory representing the virtual machine's physical memory. This memory would be at least as persistent as the virtual machine itself, and any packets transmitted via a VMNet would presumably only be interesting as long as their corresponding virtual machine exists.

The primary disadvantage of modifying the host OS is that it requires the cooperation of OS vendors, or, in the case of Linux, the active support of Linux kernel maintainers. Otherwise the optimization will not be available on unmodified off-the-shelf host OSes.

4.5 Bypassing the Host OS

As long as actual transmits to and receives from the physical network require a world switch back to the host operating system and the VMApp, an unavoidable latency and CPU overhead will remain. Additionally, the VMM will have to take native IRQs while running, world switch them back to the host, and wait for incoming packets to work through the host and VMApp before they reach the guest. *This fundamentally limits the I/O performance of a hosted virtual machine architecture.* To truly maximize I/O bandwidth, the VMM must drive the I/O device directly. The guest OS could potentially drive the device directly too, but this requires either hardware support or memory access restrictions to preserve safety and isolation.

With its own device drivers, the VMM can send and receive packets without any mandatory world switches and relay receive IRQs to the guest almost immediately. Additionally, there would be no need for a separate VMNet driver. However, adding device drivers to the VMM represents a major trade-off. Recall that VMware Workstation supports a wide variety of hardware devices because of the hosted architecture. It automatically gains support for new I/O devices and bug-fixes in existing drivers as soon as the host OS does. A VMM that requires its own NIC drivers would require an investment of resources in developing, testing, and updating its hardware support.

As described, each VMM is associated with a single virtual machine. In order to share an I/O device among several virtual machines, the VMM would have to be extended to include a global component that recognizes the individual virtual machines and their VMMs. The global component would effectively be a kernel that is specifically designed for managing VMM worlds. In addition to driving the device, the global component would have to provide software to multiplex more than one VMM onto a single I/O device. This technique is used in VMware ESX Server™, where achieving native I/O performance for high-speed devices is an important requirement.

5 Related Work

Providing interoperability and preserving compatibility are frequently necessary when introducing any new technology. As computer architectures and operating systems advance, they need to remain compatible with existing software and applications. By providing a hardware abstraction layer, virtual machine technology allows hardware differences to be hidden from legacy software, and allows multiple incompatible computing environments to co-exist on a machine.

Achieving native machine performance is a prime target of virtual machine technology. The ability to execute virtual machine code directly on the hardware allows the technology to outperform other technologies based on machine simulation or emulation. Subsequent to the early mainframe virtual machine support, IBM designed a number of architectural features to further enhance the performance of virtual machines. Gum [7] describes a number of hardware assists in the IBM System/370 architecture for further reducing the overhead of handling privileged guest instructions, guest memory address translation, and multiprocessing support.

Borden *et al.* [2] describe PR/SM, a partitioning feature on the IBM 3090 series of mainframes that allows specific devices, I/O channels and memory address ranges to be dedicated to a virtual machine. Guest I/O accesses can then be handled directly by the hardware without requiring VMM intervention. Borden *et al.* report that this feature allows a virtual machine with dedicated I/O devices to achieve within 1–2% of native hardware performance. A PC-based server platform with similar partitioning features would allow VMware's virtual machines to do the same.

Hall and Robinson [8] describe virtualizing the VAX architecture which, like the IA-32 architecture, is not naturally virtualizable and has more than two protection rings. They rely on modifications to the VAX architecture as well as the microcode. In contrast, VMware's virtualization technology does not require any hardware modifications.

Bugnion *et al.* [3] apply virtual machine technology towards providing scalable performance on large scale NUMA machines. Most commodity operating systems do not scale to a large NUMA machine without extensive modifications. However, a virtual machine monitor can be designed from the ground up to manage such a machine and hide its NUMA nature from a commodity OS. The machine can then run multiple commodity OS images, with each OS allocated as many CPUs as it can scale to.

VMware Workstation's hosted virtual machine architecture relies on user-level emulation of I/O devices. This parallels the approach taken by microkernel-based operating systems (*e.g.,* Mach [6]) which rely on user-level emulation of operating system APIs to provide multiple application environments on a single machine. The primary difference lies at the abstraction layer: while virtual machines abstract the hardware layer, microkernels abstract the OS API layer. Härtig *et al.* [9] describe techniques for improving the performance of microkernel-based systems.

6 Summary and Conclusions

This paper describes VMware's hosted virtual machine architecture as implemented in VMware Workstation. This architecture enables VMware Workstation to support a wide variety of PC hardware without special device drivers and to present a constant and hence portable virtual hardware environment. Additionally, co-existing with an commodity operating system simplifies installation and use for users and reduces the complexity of the virtual machine monitor component for the developers.

The hosted architecture splits its functionality between a VMM component that virtualizes the CPU, and a VMApp component that runs as a normal application on a host OS and handles I/O to the native devices on behalf of a virtual machine. I/O intensive workloads, in addition to running significant amounts of privileged code, require heavy-weight world switches from the VMM back to the VMApp on the host. While this is unimportant for low bandwidth devices like keyboards or mice, it can potentially prevent more demanding devices from achieving the same I/O saturation as their native counterparts. This paper focuses specifically on NIC virtualization. It presents optimizations to VMware Workstation 2.0 that allow a virtual machine hosted on a 733 MHz Pentium III CPU to saturate the network without becoming CPU bound.

The key strategy behind all the implemented optimizations is to reduce the number of world switches. The first optimization takes advantage of the fact that only a fraction of the I/O accesses to the virtual NIC causes packets to be transmitted. The remainder do not require any access to the host hardware, allowing the VMM to handle them directly instead of switching back to the host world. This optimization alone reduces CPU utilization to the point where the network link is completely saturated on a 733 MHz CPU.

The second optimization reduces the remaining world switches and trims their overhead. When the world switch rate is high enough, rather than switch back to the VMApp immediately to send each packet, the VMM gathers up to 3 packets at a time before switching back to the VMApp to send them all at once. An extra benefit of this clustering is that transmit IRQs from the native NIC becomes more likely to arrive in the host world (while sending successive packets) than in the VMM world where they would require an immediate world switch.

The third optimization uses shared memory between the VMNet driver and the VMApp to reduce the need to issue `select()` calls from the VMApp. This optimization allows the VMApp to detect which NIC IRQ requires contacting the VMNet and which NIC IRQ can immediately switch back to the VMM without spending extra time in the VMApp. Together, these three optimizations reduce the CPU utilization of the 733 MHz CPU virtual machine to around 78%. The optimizations also more than double the achievable network throughput on a 350 MHz CPU virtual machine.

The experimental results confirm that CPU overheads of a hosted virtualization strategy can prevent an I/O intensive virtual machine workload from matching the performance of the same workload on native hardware. In the straightforward implementation, frequent I/O causes frequent world switches that artificially limit the I/O utilization because the workload becomes CPU bound. However, even while remaining within a hosted virtual machine architecture, we are able to eliminate spurious world switches and even restructure around seemingly mandatory crossings with significant reduction in CPU utilization to the point that a 733 MHz Pentium III system is I/O bound with plenty of CPU cycles to spare.

CPUs are constantly getting faster and a 733 MHz Pentium III is at or below entry level for today's corporate PCs. Further, very few desktop workloads saturate a full 100 Mbit link with any regularity or frequency. Taken in conjunction with the portability, device independence, and coexistence a hosted architecture provides, VMware Workstation's achievable I/O performance strikes a good balance between performance and compatibility for its target desktop usage. The balance may change of course when gigabit networks become prevalent, depending on how fast CPUs will be by then.

Acknowledgments

VMware Workstation's hosted virtual machine architecture is the brainchild of Mendel Rosenblum, Edouard Bugnion, Scott Devine and Edward Wang. Regis Duschene provided several optimizations to the network subsystem. The anonymous referees provided useful feedback that improved the paper. Finally, this paper would not be possible without the contributions of the dedicated employees of VMware, Inc.

References

[1] AMD Corporation, Sunnyvale, CA. *Network Products: Ethernet Controllers Book 2*, 1998.

[2] Terry L. Borden, James P. Hennessy, and James W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, 1989.

[3] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[4] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[5] Robert. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[6] David Golub, Randall Dean, Allesandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the USENIX 1990 Summer Conference*, June 1990.

[7] Peter H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[8] Judith S. Hall and Paul T. Robinson. Virtualizing the VAX Architecture. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 380–389. ACM, May 1991.

[9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles*. ACM, October 1997.

[10] Intel Corporation, Santa Clara, CA. *Intel Architecture Developer's Manual. Volumes I, II and III*, 1998.

[11] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.