Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments

Hideki Eiraku, Yasushi Shinjo
Department of Computer Science
University of Tsukuba
Tsukuba, Ibaraki 305-8573, Japan

Calton Pu, Younggyun Koh
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0765 USA

Kazuhiko Kato
Department of Computer Science
University of Tsukuba
Tsukuba, Ibaraki 305-8573, Japan

ABSTRACT

This paper proposes a novel method of achieving fast networking in hosted virtual machine (VM) environments. This method, called socket-outsourcing, replaces the socket layer in a guest operating system (OS) with the socket layer of the host OS. Socket-outsourcing increases network performance by eliminating duplicate message copying in both the host OS and the guest OS. Furthermore, socket-outsourcing significantly enhances inter-VM communication within the same host OS since it enables network packets to bypass the protocol stack in guest OSes. Socket-outsourcing was implemented in two representative operating systems (Linux and NetBSD) and on two virtual machine monitors (Linux KVM and PansyVM). These virtual machine monitors provided support for socket-outsourcing through shard memory, event queues, and VM-specific Remote Procedure Call between a guest OS and a host OS. The experimental results revealed that a guest OS outsourcing the socket layer achieved the same network throughput as a native OS using up to four Gigabit Ethernet links. Moreover, the benchmark results obtained from an N-tier Web application that generated a significant amount of inter-VM communication indicated that socket-outsourcing improved performance by up to 45 percent compared with conventional hosted VM environments.

Categories and Subject Descriptors D.4.4 [Operating Systems]: Communications management

General Terms Performance, design

Keywords Virtualization, hosted virtual machine monitors, host operating systems, guest operating systems, paravirtualization, outsourcing, socket API

1. INTRODUCTION

Virtual Machine Monitors (VMMs) provide significant advantages in terms of isolation and portability of applications. An early classification [8] of VMMs has divided them into two types: Type I VMMs, which are hypervisor-based VMMs running on bare hardware such as Xen [9] and VMware ESX Server [25], and Type II VMMs (also known as hosted VMMs), such as VMware Workstation [23], Linux KVM [13], and User Mode Linux (UML) [4]. Compared to Type I VMMs, hosted VMMs have advantages such as host operating system (OS) reuse, and OS installation as a normal application program [20], but hosted VMMs incur a relatively high performance penalty, especially in I/O processing.

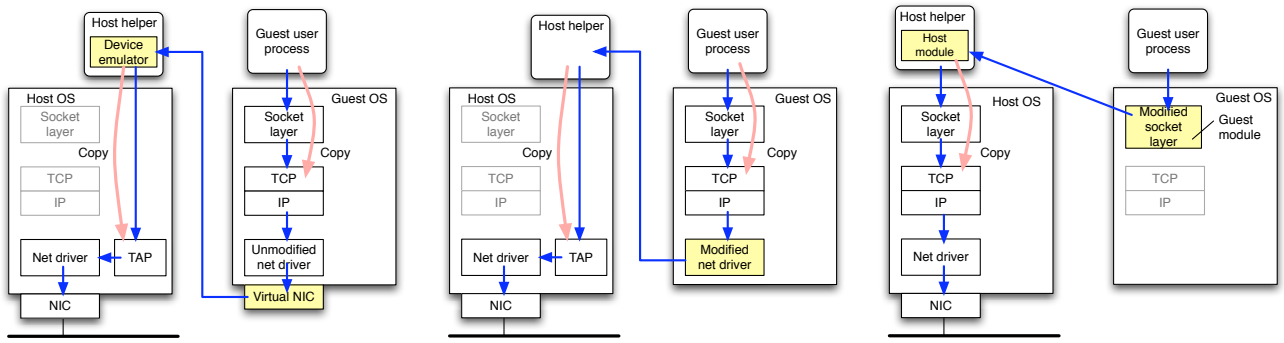
Compared to native operating systems (OSes), there are four main sources of additional overhead in a guest OS running on a hosted VMM: (1) heavy costs to capture CPU exceptions including system calls and page faults, (2) execution of privileged functions in the guest OS kernel in user mode, (3) duplicated functionality between a guest OS and a host OS in I/O processing such as network protocol stacks, and (4) redundant copying of buffers across multiple user-kernel boundaries. Recent hardware support for virtualization, such as Intel Virtualization Technology (VT) and AMD Virtualization (AMD-V), have helped to reduce or remove sources (1) and (2) of the performance penalty. However, due to the architecture of hosted VMMs and "inherent" duplication of functionality between a guest OS and a host OS, sources (3) and (4) of the performance penalty constitute serious research challenges that have contributed to the slow adoption of hosted VMMs. While some advanced hardware, such as Intel VT for Directed I/O (VT-d) has helped to remove these performance penalty sources in Type-I VMMs, it is hard to use such hardware in Type II VMMs.

In a similar way to optimizing hypervisors (the lower layer in Type I VMMs), optimizing hosted VMMs has focused on bypassing the layers in the host OS (the lower layer in hosted VMMs). For example, Virtio in Linux helps to link a specialized guest OS network driver to a specialized host OS network driver to avoid redundant protocol processing and buffer copying in the host OS [21]. While this effectively eliminates some of the previously mentioned cost factors, this hosted VMM analog of paravirtualization is unable to avoid several sources that incur a performance penalty, including:

- Duplicate message copying in both the host OS and the guest OS.
- The high overhead in inter-VM communication. For example, two guest OSes on the same host OS need to go through full network protocol stacks.

The main contribution of this paper is that it presents an alternative approach to optimizing hosted VMMs, called *outsourcing*. In contrast to paravirtualization, which optimizes (low-level modules of) the guest OS to communicate with the hypervisor, outsourcing specializes (high-level modules of) the guest OS to communicate with high-level facilities of the host OS. Specifically, the outsourcing of the socket layer is called *socket-*

SAC'09, March 8-12, 2009, Honolulu, Hawaii, U.S.A.



(a) With device emulator based on full virtualization.

(b) With modified guest device driver based on paravirtualization.

(c) With modified socket layer and host module based on outsourcing.

Figure 1 Network I/O methods in hosted VMM.

outsourcing. As an illustrative example, Linux Virtio helps to bypass the host OS protocol stack by invoking a low-level host driver from the guest OS. In contrast, socket-outsourcing bypasses the guest OS protocol stack by invoking the socket layer in the host OS. This design eliminates duplicate message copying and reduces the inter-VM communication overhead.

We implemented socket-outsourcing in two representative guest OSes (Linux and NetBSD) running on two hosted VMM environments (Linux KVM [13] and PansyVM). PansyVM is a successor to LilyVM [5], a VMM for the x86 architecture based on paravirtualization with static code rewriting. Our experiments revealed that guest OSes using socket-outsourcing can achieve the same network throughput as a native OS using up to four Gigabit Ethernet links. Using an e-commerce benchmark (RUBiS) that performed significant inter-VM communication in a consolidated server environment, socket-outsourcing improved performance by up to 45 percent compared with conventional hosted VM environments.

The remainder of the paper is organized as follows. Section 2 compares the network I/O design choices for VMMs. Section 3 discusses issues with application compatibility and IP addresses in socket-outsourcing. Section 4 presents the VMM support required for outsourcing and the interface between the guest OS and the host OS for socket-outsourcing. Section 5 explains the mapping of the socket layer in Linux and NetBSD onto the host OS. Section 6 describes the implementations of the host-side modules in socket-outsourcing and the VMM support for outsourcing in Linux KVM and PansyVM. Section 7 presents the experimental results. Section 8 covers related work. Finally, Section 9 concludes the paper.

2. Network I/O in VMMs

Sections 2.1 and 2.2 outline the performance problems encountered by conventional network I/O processing in hosted VMM environments. We then describe socket-outsourcing and the approach we used to address these performance problems.

2.1 Full Virtualization Through Device Emulation

Figure 1(a) shows network I/O with a device emulator to achieve full virtualization. The guest OS includes a native device driver for a popular network device, e.g., NE2000 and RTL8139, since there are no standards such as SCSI and ATA for networking. The underlying VMM provides an emulator for these popular network devices. When the guest device driver executes an I/O instruction, the VMM traps the execution and emulates it on behalf of the hardware. Although full virtualization has good compatibility (no changes to the guest OS), there are some well-known performance problems due to emulation of devices by the software [23][24].

2.2 Hosted VMM Analog of Paravirtualization

Figure 1(b) outlines the network I/O processing in hosted VMM through an approach similar to paravirtualization, used in Xen, Linux KVM with Virtio support, and User Mode Linux. In this method, the guest OS uses a special *paravirtual* device driver that communicates with a low-level network module running in the host OS, such as a backend driver in Xen and a TUN/TAP driver in Linux.

Paravirtualization achieves better performance than full virtualization, but two problems still remain. First, it is hard to omit duplicate message copying in both the guest OS and the host OS. For example, let us assume that a guest process sends a message with the TCP in Figure 1(b). The guest OS must perform the first copying for retransmission due to packet losses. The host OS must perform the second copying to allow the application to fill the buffer with the next message. The second problem involves high overhead in inter-VM communications. Message exchanges between two guest OSes in the same host OS require processing by two full protocol stacks and a software switch module.

2.3 Overview of Socket-Outsourcing

To mitigate the performance problems with paravirtualization, we propose a new network I/O method, i.e., socket-outsourcing. Figure 1(c) illustrates the control flow for network I/O processing in socket-outsourcing. Unlike paravirtualization, which attempts to bypass redundant processing by using low-level interfaces (e.g., device drivers), outsourcing attempts to bypass redundant processing by using a high-level interface (socket). Outsourcing replaces a high-level module in the host OS, which is referred to as *a guest module*, with one that is specialized. In Figure 1(c), the socket layer is a guest module in outsourcing and it is modified as the device driver is modified in the paravirtualization in Figure 1(b). The modified socket layer communicates with a program called *a host module*. The host module receives requests from the

guest module and issues system calls to the host OS through a standard API. In Figure 1(c), the host module runs in a user-level process. We can also execute the host module in the kernel.

Socket-outsourcing has two performance advantages against paravirtualization. First, we can omit message copying in the guest OS. We will describe this in more detail in Section 5.2. Second, we can accelerate inter-VM communication. If a guest process running in a VM sends a TCP message to another process running in another VM, only the host stack handles this message, and no lower layers, such as device drivers and emulated switch devices relay the message.

3. Issues with Socket-outsourcing

3.1 Application Compatibilities

Socket-outsourcing exploits the standard socket API that both the guest OS and the host OS provide. If an application relies on non-standard implementation-specific features of the guest OS protocol stack, such an application will not work.

To mitigate this compatibility problem, we provide global and socket options. The global option controls whether or not the kernel is allowed to use the host stack by default. The socket option specifies each socket instance that can use or not use the host stack. When we are not permitted to use the host stack for a socket, we fall back to the conventional paravirtualization method.

3.2 Sharing of IP Addresses with Host OS

Simple socket-outsourcing appears to be like the network address translation (NAT) mode of regular hosted virtual machines. This means the guest OS shares the same IP addresses with the host OS.

Occasionally, we need to allocate one or more dedicate IP addresses to each VM instance. To accomplish this, we add these IP addresses to network interfaces in the host in advance. When a guest process creates a server socket and assigns the IP address with system call `bind()`, we enforce the address by restricting the arguments of system call `bind()`. When a guest process initiates a network connection as a client, we enforce the source IP address with system call `bind()` in the host module even though the guest process does not issue `bind()` in the guest OS.

4. VMM Support for Outsourcing

In paravirtualization, Xen provides shared memory and event channels for communication between frontend and backend drivers. Virtio-enabled KVM provides similar facilities. In outsourcing, we provide Remote Procedure Call (RPC) as well as shared memory and event queues.

4.1 Communications for Outsourcing

In outsourcing, the VMM must provide communication and synchronization facilities between a guest module and a host module. Figure 2 shows three main facilities of the VMM:

Shared memory: The guest module allows the host module to access its memory regions.

Event queues: An event queue is a data structure allocated in the shared memory. This facility is used for asynchronous communication between the host module and the guest module.

VM Remote Procedure Call (VRPC): The guest module calls the host module and blocks until the host module returns a reply.

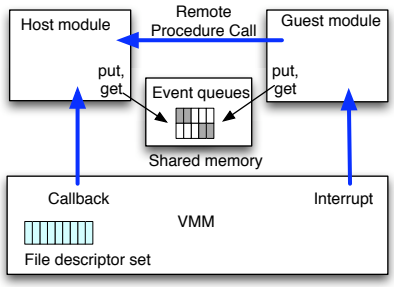


Figure 2 VMM support for communication between guest and host modules.

In addition to these communication facilities, the VMM maintains a file descriptor set (*FD set*). The FD set is similar to the `fd_set` type of system call `select()`. When the VMM notices status changes in files in the FD set, it calls back the host module. Since the VMM must handle other events such as timer interrupts and console I/O, the VMM manages all file descriptors in a centralized way, and notifies each module of status changes in the module's files. On the guest OS side, the VMM provides the facility to generate interrupts to notify the guest OS that events have arrived. Generating interrupts is a common facility of the VMM.

VRPC is an RPC facility with following specialized features for the hosted VMM environment. First, the server should not block. For example, to receive a message, the server should return an error immediately when no message has arrived. Second, VRPC parameters are passed via the shared memory and no marshaling is needed. Third, VRPC does not have to handle errors such as when the server is down and the network is disconnected. These features simplify the implementation of outsourcing. The details on implementing VRPC are described in Section 5.2.

4.2 VRPC Interfaces

Table 1 summarizes the interface between the host module and the guest module for socket-outsourcing. The first four procedures initialize and finalize the module and instances.

The procedures from `h_connect()` to `h_getsockopt()` provide similar functions in the regular socket API except that these procedures never block. For example, procedure `h_recvmsg()`, which receives a message from a socket, immediately returns an error when no message has arrived. The last procedure `h_getstatus()` returns the status of a socket, and is used to implement I/O multiplexing, such as system calls `select()` and `poll()`. Another difference is in the identifier of sockets. These procedures in Table 1 take a handle that is returned by procedure `h_newsocket()`.

4.3 Event Interfaces

The host module uses several kinds of events in Table 2 to notify the guest module of actions of interest. For example, when the host module notices that a socket has an incoming message, it sends event `ARRIVED` to the guest module. In implementing socket-outsourcing, the event queue is only used in one direction; the host module sends events to the guest module, but not vice versa.

5. Mapping Guest OS onto Host OS

We demonstrate the practicality and efficiency of the socket-outsourcing approach by implementing it in two representative guest OSes: Linux and NetBSD. Since the Socket API has been designed to add new protocols in modular way, we were able to implement socket-outsourcing in these two OSes by replacing corresponding modules.

5.1 Socket-Outsourcing Implementation in Linux

The socket layer of Linux allocates socket objects that export functions described in the structure, proto_ops. To implement socket-outsourcing in Linux, we replaced functions in structure proto_ops for TCP and UDP with substitute functions.

In this section, function inet_recvmsg() is used as an example to illustrate the key idea behind implementing socket-outsourcing. Function inet_recvmsg() is called from not only system call recvmsg() but also system calls recv(), recvfrom(), read(), and readv() to receive a TCP message.

Figure 3 shows the algorithm for function vinet_recvmsg(), which is a substitute function for inet_recvmsg(). First, this function allocates non-pageable memory in the kernel space. Next, it performs a VRPC to the host module. If a message has arrived, the VRPC returns the number of bytes received. In this case, the function copies out the message to the user space, frees the non-pageable memory, and returns the same value. We will discuss how this copying can be avoided in Section 5.2.

If no message has arrived at the socket, the current process blocks and waits for a new message. When the host module notices a message has arrived, it inserts an event into the queue for the guest module, and asks the VMM to generate an interrupt to the guest OS. The interrupt handler of the guest OS receives the event, and unblocks the waiting process. When the process becomes ready again, it tries the VRPC again to obtain the received message.

To implement socket-outsourcing in Linux, we added 700 lines of code to Linux 2.4.27, and 1300 lines of code to Linux 2.6.25.

Table 1 VRPC interface for socket-outsourcing.

Names	Descriptions
h_init	Initialize the module.
h_final	Finalize the module.
h_newsocket	Create a socket instance.
h_delete	Delete a socket instance.
h_bind	Bind a name to a socket.
h_listen	Listen for connections on a socket.
h_connect	Initiate a connection on a socket.
h_accept	Accept a connection on a socket.
h_sendmsg	Send a message from a socket.
h_recvmsg	Receive a message from a socket.
h_shutdown	Shutdown part of a full-duplex connection.
h_getsockname	Get the name of a socket.
h_getpeername	Get the name of a connected peer.
h_setsockopt	Set options on a socket.
h_getsockopt	Get options on a socket.
h_getstatus	Get the status of a socket.

1. Allocate non-pageable memory.
2. Call host procedure h_recvmsg() with the address of the non-pageable memory.
3. If the procedure returns a no-message error, block the current process. When the current process is unblocked, go to 2.
4. Otherwise, copy the received message from non-pageable memory to the destination memory in the user process.
5. Free the non-pageable memory, and return the result (the number of bytes received or an error) to the user.

Figure 3 Algorithm for receiving TCP message in Linux based on socket-outsourcing.

5.2 Optimistic Copy Avoidance in Linux

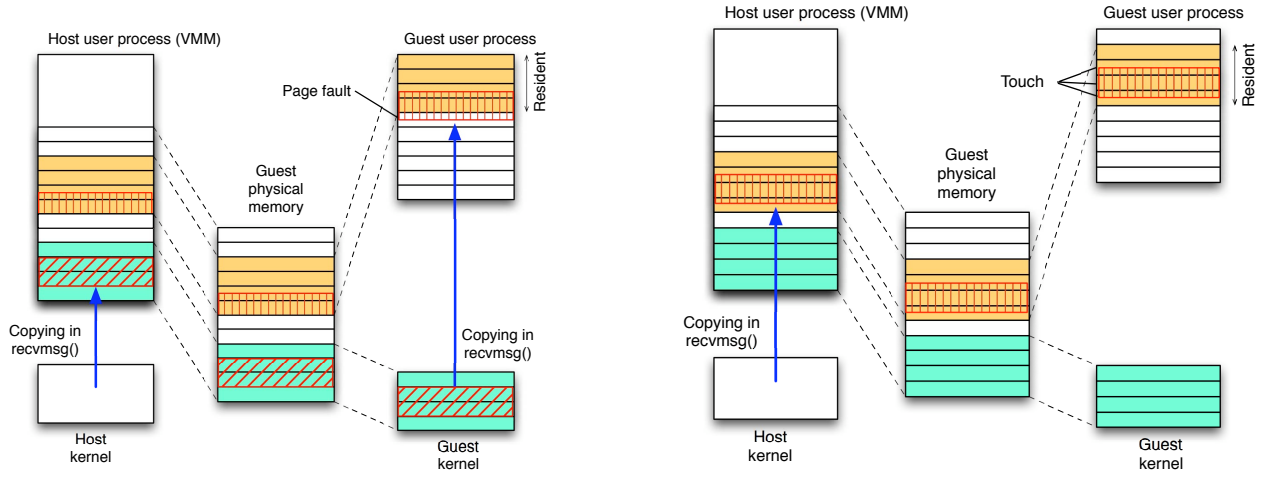
In the previous subsection, we discussed that an arriving message is copied twice: from the host kernel to the host user process and from the guest kernel to the guest user process. The host user process and the guest kernel use shared memory mapping (Figure 4(a)) to avoid the third copying. First, the VMM allocates *physical memory* for the guest OS at the time of initialization. Second, the same memory is mapped to the logical address space of the host process on which the guest OS resides. When the host module in this process is called through procedure h_recvmsg(), the host module first extracts the parameters and obtains the destination address in the guest logical address. The host module translates the guest logical address into the guest physical address using the page table of the VMM. This translation never fails because the destination is fixed non-pageable memory in the guest kernel. Next, the host module translates the guest physical address to the host logical address. Finally, the host module issues system call recvmsg() with the host logical address. The host OS performs the first copying in system call recvmsg(). After VRPC h_recvmsg() returns from the host OS, the guest OS performs the second copying from the kernel non-pageable memory to the user memory.

Figure 4(b) shows how we can avoid performing the second copying, but a page fault can still occur in the guest OS. In Figure 4(a), the destination of recvmsg() in the guest OS occupies three pages: two pages are resident in the main memory, and the last page is not resident at the guest OS level. In this case, the host module is unable to translate the guest logical address of the last page into the guest physical address.

Regular OS kernels provide powerful copying functions to take care of page faults, e.g., copy_to_user() in Linux and copyout() in BSD. If a page fault occurs, the page fault handler first allocates a memory page, and resumes copying. In outsourcing, we took an

Table 2 Events from host module to guest module for socket-outsourcing.

Names	Descriptions
ESTABLISHED	A connection has been established.
EMPTY	The send buffer becomes available.
ARRIVED	A message has arrived.
OOB_ARRIVED	An out-of-bound message has arrived.
ERROR	An error occurred.



(a) Duplicate copying in recvmsg(). (b) Single copying in recvmsg().

Figure 4 Avoiding copying in socket-outsourcing.

1. Touch the destination memory in the user process with the write mode.
2. Call host procedure h_recvmsg() with the address of the destination memory in the user process.
4. If the procedure returns a no-message error, block the current process. When the current process is unblocked, go to 1.
5. If a page fault occurs in the host module, fall back to the copying method in Figure 3.
6. Otherwise, return the result (the number of bytes received or an error) to the user.

Figure 5 Algorithm for receiving TCP message with optimistic copy avoidance.

optimistic approach to solving this page-fault problem.

Figure 5 shows the algorithm for function vinet_recvmsg() with optimistic copy avoidance. First, the function touches each memory page (by reading/writing one byte) of the destination to retrieve the pages into physical memory. In Linux, touching is implemented by calling functions get_user() and put_user(). Next, the function performs a VRPC to receive a message with the logical address of the user process. If the host module returns a page-fault error, the function falls back to the copying method. If the host module returns the number of bytes received, function vinet_recvmsg() returns the same value to the user process.

5.3 Socket-outsourcing Implementation in NetBSD

NetBSD has a TCP/IP stack that is derived from 4.4 BSD [15]. Unlike Linux, the protocol stack of 4.4 BSD consists of generic functions that can handle several protocols including TCP, UDP, Unix Domain Sockets, and Unix pipes. These generic functions call function usrreq() (user request) when they need protocol specific actions. For example, system call bind() calls usrreq() with request PRU_BIND. Each protocol has its own usrreq() function. Therefore, we replaced function usrreq() for TCP and UDP to implement socket-outsourcing. In addition to usrreq(), we had to override function soreceive() to avoid copying as described in Section 5.2.

In summary, we added 600 lines of code to NetBSD 2.0, and 1000 lines of code to NetBSD 4.0 to implement socket-outsourcing.

6. Implementation of Host Modules and VMM Extensions

6.1 User-level Host Module for Socket-Outsourcing

We implemented a user-level host module for socket-outsourcing. This module runs in the user-level VMM, acts as a VRPC server for the guest module and provides the VRPC interface described in Table 1. Most procedures in Table 1 issue corresponding system calls for the host OS. For example, procedure h_connect() issues system call connect().

Several actions occur when the host module receives request h_newsocket(). First, the host module issues system call socket() to the host OS. Second, the host module provides a non-blocking I/O feature to the new socket by using the fcntl() system call with the parameter, O_NONBLOCK. This is essential to achieve the non-blocking feature described in Section 4.1. Third, the host module registers the file descriptor of the new socket to the FD set in the VMM. After this, the VMM calls the host module back when some status in the socket changes. Finally, the host module returns a handle for the socket.

When the VMM notices a change in status, such as a message arriving, the VMM calls the host module back. The host module analyzes the status change, and usually sends an event to the guest module through the event queue. For example, when the host module notices that a socket has an incoming message, the module sends an event, ARRIVED in Table 2, to the guest module. Finally, the host module asks the VMM to generate an interrupt to the guest OS to deliver the event.

6.2 Extending Linux KVM

The Linux Kernel-based Virtualization Driver (KVM) is a kernel extension (a pseudo-device driver) that provides a framework for writing a VMM at the user-level [13]. KVM captures the execution of privileged instructions and sensitive non-privileged instructions by using hardware support (Intel VT or AMD-V). KVM gives notifications of the executions to a user-level program

through system call ioctl(). The distribution of KVM includes a modified QEMU [2] for emulating I/O devices.

We have extended KVM to provide VMM-support facilities including shared memory, event queues, and VRPC, as described in Section 4.1. In KVM, the user-level QEMU code can access any guest physical memory. We used this feature as shared memory between the host and guest modules. In shared memory, we provided library functions to manipulate queues for both the host and guest modules.

We implemented VRPC using the instruction vmcall in Intel VT-enabled CPUs. The guest-side code first places the VRPC parameters in the stack and registers. Next, it executes the vmcall instruction. Executing this instruction causes a trap to the VMM, also known as a VM exit in Intel's terminology. The modified kernel module of KVM transfers the flow of control to the user-level QEMU code. The modified QEMU analyzes the reason for the trap, and calls the host module.

6.3 PansyVM

PansyVM is a successor to LilyVM [5], a VMM for the x86 architecture based on paravirtualization. In LilyVM and PansyVM, the sensitive instructions [17] of x86 are translated into library function calls at the time of compilation. This translation reduces the porting efforts of the guest OS in paravirtualization. It also enables a guest OS to be executed on a CPU that does not have VT capabilities. The previous version of PansyVM used paravirtual drivers for networks and block devices for performance. Unlike LilyVM, PansyVM includes a kernel-level code for fast handling of exceptions.

We implemented shared memory, event queues, and VRPC for PansyVM in a similar fashion to their implementations in KVM. Since PansyVM does not require VT capabilities, we implemented the VRPC facility by extending the regular hypervisor call mechanism.

7. Evaluations

7.1 Experimental Setup

We used three Intel Xeon 5160 3.0-GHz machines (Dell Power Edge 1900) with 4 MB of L2 cache and 2 GB of main memory for our experiments. Each machine had four network interface cards (NICs), all connected to a gigabit network switch (Nortel 3510-24T). We turned off the machine's SMP capabilities to reduce the variance and increase the reproducibility of the measurements (CPU overheads of target VMM environments).

We ran the experiments on two VMM environments: KVM 66, and PansyVM 2008-05-05. Since KVM uses QEMU's device emulation modules, we used QEMU's network device i82557b in the emulation method.

We conducted all experiments using Linux 2.6.25 for the guest and host OSes in both virtual environments. We used a disk partition as the backing store of a guest disk image. We set the main memory of the guest Linux to 256 MB while the host Linux was allowed to use all 2 GB of main memory.

In Linux KVM, we compared the emulation method (KVM-emu), the paravirtualization method (KVM-virtio), and our outsourcing method (KVM-out). In PansyVM, we compared the paravirtualization method (PansyVM-tap) and the outsourcing method (PansyVM-out). We used a TAP device for the paravirtualization in PansyVM.

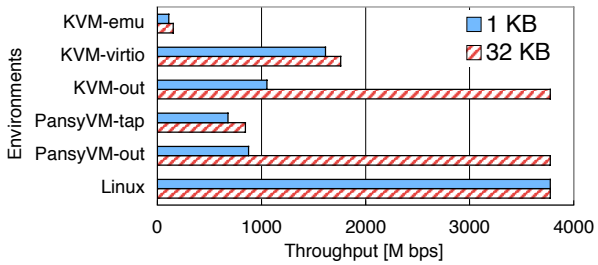


Figure 6 Maximum TCP throughput measured with iperf (sending).

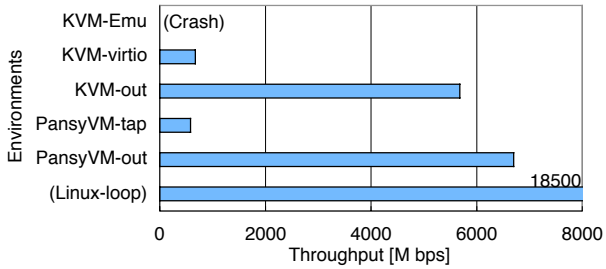


Figure 7 Maximum inter-VM TCP throughput within single host.

In the experiments that followed, we measured the performance of the guest OS running in each virtual environment as well as the performance of a native OS running on the non-virtualized environment.

7.2 Maximum Network Throughput

We used iperf [7], a tool for measuring network bandwidth, to measure the maximum network throughput between our experimental machines. Since each machine had multiple NICs, we launched multiple instances of iperf for each NIC simultaneously and calculated the combined network throughput by adding the measured bandwidth of each NIC. The iperf messages we used for all the experiments were 1 and 32 KB in size. The MTU of each NIC was 1500 B.

Figure 6 shows our measured results when we increased the number of NICs up to four. Although we measured both the sending and receiving performance, we included the results of sending in Figure 6 because both sending and receiving yielded similar results. Sending was faster than receiving since sending consumed less CPU resources. Although PansyVM was faster than KVM by using file I/O benchmarks, PansyVM was slower than KVM by using this network I/O benchmark. This is because the paravirtual driver of PansyVM required an extra message copying.

KVM-out and PansyVM-out sped up TCP sending throughput by 25 times compared with the emulation method, and reached native performance when the message size was 32 KB. This improved performance with outsourcing was mainly achieved by eliminating the copying as described in Section 5.2. When the message was 1 KB, KVM-virtio was better than KVM-out because there were numerous host-guest switches. In KVM-virtio, the paravirtual driver queued an I/O request for each network frame with a Maximum Transfer Unit (MTU), and several

requests were batched and processed in a single host-guest context switch. In KVM-out, the guest OS interacted with the host OS for each original message.

7.3 Inter-VM Communication

We measured the maximum TCP throughput between two virtual machines running on the same host OS by using the iperf tool as discussed in Section 7.2. The bar chart in Figure 7 shows the results. The last Linux-loop means communication between two processes within a host OS via the local loopback interface. We were not able to obtain results for KVM-emu because flooding messages due to iperf crashed the VM.

As seen in this chart, KVM-out achieved 5700 Mbps and PansyVM-out achieved 6700 Mbps and they were faster than KVM-virtio and PansyVM-tap. KVM-virtio was slow in our experiments due to a scheduling problem. In KVM-virtio, the round trip time (RTT) of the ping message was 20 ms, which was double that of a periodic timer interval.

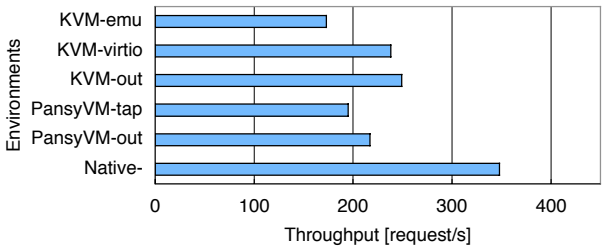
7.4 Web Application Benchmark

To evaluate the impact of performance on applications, we measured throughput with the RUBiS benchmark [3], consisting of 26 interactions with a Java-based auction site running Web servers, application servers, and database servers. Examples of RUBiS transactions include: login, browsing, searching, purchasing, and selling. We used the servlet version of RUBiS, consisting of servlets running in Tomcat, a database server (MySQL), and a client emulator. We ran Apache Tomcat and MySQL servers in a single virtual environment, or in two dedicated virtual environments, and executed the client emulator on the other machine running native Linux. To measure the best throughputs, we changed the number of clients within a range from 200 to 2000. We used the default workload of RUBiS 1.4.3, and initialized the database with a 30-MB set for each execution. We ran Apache Tomcat 6.0.16 by Java EE SDK 5.04 and MySQL 5.0.

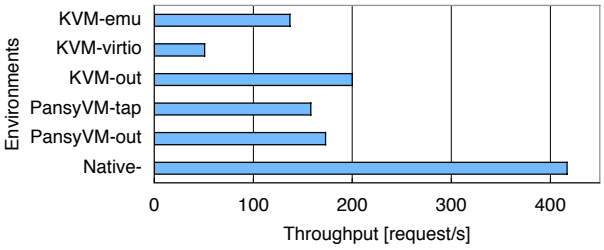
The bar charts in Figure 8 show the RUBiS throughputs in various virtual environments including native Linux. When we ran both Tomcat and MySQL in a single virtual machine (Figure 8(a)), KVM-out sped up the throughput by 44 percent and PansyVM-out by 25 percent compared with KVM-emu, and these were faster than the paravirtual throughputs. In this experiment, *a localhost optimization* in KVM-out and PansyVM-out worked well. In this optimization, Tomcat and MySQL communicated within the guest OS as in KVM-emu and KVM-virtio.

Using two virtual machines (Figure 8(b)), KVM-out sped up the throughputs by 46 percent and PansyVM-out by 26 percent compared with KVM-emu, and these were faster than the paravirtual throughputs. In this experiment, KVM was faster than PansyVM because KVM effectively used hardware support for virtualization.

In summary, socket-outsourcing improved the performance of a Web application with databases because it accelerated both the communication to clients and the communication between the Web server and database server.



(a) Using single virtual machine



(b) Using two virtual machines

Figure 8 Results of RUBiS benchmark.

8. Related Work

Xen [9] avoids heavy device emulation by providing a paravirtual network driver based on paravirtualization. In Menon et al. [16], the guest OS running in a Domain-U could use intelligent NIC facilities, including scatter/gather I/O and TCP/IP checksum offloading. The Linux kernel after version 2.6.24 includes a framework called Virtio for paravirtual device drivers [21]. The VMware Virtual Machine Interface (VMI) provides I/O facilities for paravirtual network drivers in the VMware Workstation [24]. These approaches focus on low-level modules based on paravirtualization while we focused on a high-level module based on outsourcing.

XWay [10], XenLoop [26], and XenSocket [27] accelerate inter-VM communication in Xen by using shared memory and other communication support by Xen. These are dedicated for inter-VM communication. Socket-outsourcing improves performance not only for inter-VM communication but also for outside communication to the Internet.

Outsourcing the socket layer resembles TCP/IP offloading [6][11][18][19]. Outsourcing is different in that it delegates the facility of a module to another software module while TCP/IP offloading delegates the facility to hardware. SOP modifies the socket layer for high-level TCP/IP offloading engines (TOEs) [22]. SOP does not support virtual environments, and no results for performance have been reported.

The network performance of hosted VMMs has been improved by reducing the number of context switches, avoiding copying, and specializing network stacks [12][14]. While these approaches have focused on optimizing and specializing guest OSes, socket-outsourcing improves network performance by bypassing the entire processing in guest OSes.

User Mode Linux is a port of Linux to Linux [4], and includes a special file system called hostfs to access the files in the host OS. Cooperative Linux (coLinux) is a port of Linux to Windows as well as Linux [1]. Cooperative Linux includes a special file system called cofs to access the Windows files. Both hostfs and cofs can be regarded as outsourcing of the VFS layer. This paper has discussed outsourcing of the socket layer.

9. Conclusion

We described outsourcing, an approach to bypassing unnecessary overhead at a high level of abstraction for Type II VMMs (hosted VMMs). We illustrated the outsourcing approach with an experimental implementation of network I/O at the socket layer (socket-outsourcing) in two guest OSes (Linux and NetBSD) on two VMMs (Linux KVM and PansyVM). In socket-outsourcing, we modified the socket layer of the guest OS and connected it to a module running in the host OS. The guest socket layer and the host module communicate by using VMM support for socket-outsourcing, such as shared memory, event queues, and RPC specialized for VMM environments (VRPC).

The experimental measurements demonstrated socket-outsourcing significantly improved network performance in two virtual environments: Linux KVM and PansyVM. With outsourcing, Linux on PansyVM achieved native performance for both sending and receiving and Linux on KVM achieved that for sending when the message was 32 KB. Using the RUBiS e-commerce benchmark that performs significant inter-VM communications, Linux using socket-outsourcing on KVM ran faster by 45 percent and PansyVM by 25 percent than Linux on KVM using the emulation method.

References

[1] Dan Aloni: "Cooperative Linux", the Ottawa Linux Symposium (OLS-2004), pp.23-31 (2004).

[2] Fabrice Bellard: "QEMU, a Fast and Portable Dynamic Translator", the USENIX 2005 Annual Technical Conference, FREENIX Track, pp. 41-46 (2005).

[3] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel: "Performance and scalability of EJB applications", Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 246-261 (2002).

[4] Jeff Dike: "A user-mode port of the Linux kernel", 4th Annual Linux Showcase & Conference (2000).

[5] Hideki Eiraku and Yasushi Shinjo: "Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions", the USENIX BSDCon 2003 Conference (BSDCon'03), pp. 91-102 (Sep. 2003).

[6] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey: "Server Network Scalability and TCP Offload", the USENIX Annual Technical Conference, pp. 209-222 (2005).

[7] Mark Gates, Ajay Tirumala, Jon Dugan, and Kevin Gibbs: "Iperf User Docs" (2003). http://sourceforge.net/projects/iperf

[8] Robert P. Goldberg: "Survey of Virtual Machine Research. IEEE Computer, pp. 34-45 (1974).

[9] Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield: "Xen and the Art of Virtualization", 19th ACM Symposium on Operating Systems Principles, pp. 164-177 (2003).

[10] Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim: "Inter-domain socket communications supporting high performance and full binary compatibility on Xen", International Conference on Virtual Execution Environments (VEE-08), pp. 11-20 (2008).

[11] Krishna Kant: "TCP Offload Performance for Front-End Servers", IEEE Global Telecommunications Conference (GLOBECOM 03), pp.3242-324 (2003).

[12] Samuel King, George Dunlop, and Peter: "Operating System Support for Virtual Machines", the USENIX Annual Technical Conference (2003).

[13] Avi Kivity, Yaniv Kamay, and Dor Laor: "kvm: the Linux Virtual Machine Monitor", the Linux Symposium, pp. 225-230 (2007).

[14] Younggyun Koh, Calton Pu, Sapan Bhatia, and Charles Consel: "Efficient Packet Processing in User-Level OSes: A Study of UML", the 31st IEEE Conference on Local Computer Networks (2006).

[15] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman: "The Design and Implementation of the 4.4 BSD Operating System", Addison Wesley (1996).

[16] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel: "Optimizing Network Virtualization in Xen", the 2006 USENIX Annual Technical Conference, pp. 15-28 (2006).

[17] John Scott Robin and Cynthia E. Irvine: "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", the USENIX Security Symposium (2000).

[18] Murali Rangarajan, Aniruddha Bohra, Kalpana Banerjee, Enrique V. Carrera, Ricardo Bianchini, Liviu Iftode, and Willy Zwaenepoel: "TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance", Computer Science Department, Rutgers University, Technical Report DCR-TR-48 (2002).

[19] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong: "TCP Onloading for Data Center Servers", IEEE Computer, pp. 48-58 (2004).

[20] Mendel Rosenblum and Tal Garfinkel: "Virtual Machine Monitors: Current Technology and Future Trends", IEEE Computer, Vol. 38, No. 5 pp. 39-47 (May 2005).

[21] Rusty Russell: "Virtio: towards a de-facto standard for virtual I/O devices", ACM SIGOPS Operating Systems Review, Vol.42, No.95-103 (2008).

[22] Sunghoon Son, Jaeyol Kim, Eunji Lim, and Sungin Jung: "SOP: A Socket Interface for TOEs", Internet and Multimedia Systems and Applications, pp. 294-299 (2004).

[23] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim: "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", the 2001 USENIX Annual Technical Conference (2001).

[24] "Paravirtualization API Version 2.0", VMware, http://www.vmware.com (2006).

[25] Carl A. Waldspurger: "Memory Resource Management in VMware ESX Server", 5th Symposium on Operating Systems Design and Implementation (OSDI-2002), pp.181-194 (2002).

[26] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan: "XenLoop: A Transparent High Performance Inter-VM Network Loopback", International Symposium on High Performance Distributed Computing, pp.109-118 (2008).

[27] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin: "XenSocket: A High-Throughput Interdomain Transport for Virtual Machines", International Middleware Conference (2007).