KERNEL SERVICE OUTSOURCING: AN APPROACH TO IMPROVE PERFORMANCE AND RELIABILITY OF VIRTUALIZED SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

Younggyun Koh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2010

KERNEL SERVICE OUTSOURCING: AN APPROACH TO IMPROVE PERFORMANCE AND RELIABILITY OF VIRTUALIZED SYSTEMS

Approved by:

Dr. Calton Pu, Advisor
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Jinpeng Wei
School of Computing and
Information Sciences
Florida International University

Dr. Ling Liu
College of Computing
Georgia Institute of Technology

Date Approved: June 22, 2010

To My Father

ACKNOWLEDGEMENTS

First and foremost, I must express my utmost gratitude to my academic advisor, Dr. Calton Pu, who throughput my Ph.D. program has enlightened and inspired me with his wise counsel and encouragement. Through his guidance, I have learned how to develop my ideas and mature as a researcher. I am truly grateful for his support and faith in me at every step of the way in my program.

I also am deeply indebted to Dr. Yasushi Shinjo. Every time technical difficulties impeded my progress, he guided me past them with his insightful comments and suggestions. I also thank his students, Eiraku Hideki and Go Saito, for their collaboration efforts.

I also thank Dr. Ling Liu for her valuable comments on my thesis as a member of my committee. The energy and passion for research that she expressed in our group seminars always motivated me. I want to thank my thesis committee members, Dr. Mustaque Ahamad, Dr. Karsten Schwan, and Dr. Jinpeng Wei for their constructive suggestions that resulted in further improvements to my thesis. I also thank Dr. Charles Consel and Dr. Sapan Bhatia for their advice that made possible my first research paper.

During my stay at Georgia Tech, I shared office space with bright and hard-working labmates, Lenin Singaravelu, Jinpeng Wei, Qingyang Wang, Junhee Park, Galen Swint, Steve Webb, Xiong Li, David Buttler, Lakshmish Ramaswamy, and Gueyoung Jung. I have really enjoyed our spontaneous lunch outings and our discussions of our different cultures.

I was able to experience Southern warmth with Young and Moon Han and Chris and Nui Mapes, from whom I learned a great deal about American culture. We spent numerous special occasions such as Thanksgiving Day and New Year Eve's together for many years. I cherish our shared moments.

With help from my Korean friends, Minho Sung, Jaeil Choi, Wooyong Lee, Kyuhan Kim, Eunjung Kang, Sungkeun Park, Junsuk Shin, and Dongshin Kim, I was able to settle down smoothly in a totally new place, thousands of miles away from my home, without suffering any homesickness. I sincerely enjoyed the delightful conversations over countless meals I have had with Minsung Jang, Sungwon Kang, Hyojoon Kim, Jungju Oh, Moonkyung Ryu, Ja-young Sung, Jaewook Yoo, and especially Sangmin Park. I thank them for their companionship during my last days at Georgia Tech.

My old friends, Soowook Ahn, Banghoon Kim, Hunjoo Lee, Dongwon Lim, Jaein Ko, Jungtai Kim, Jaeyong Kim, Chulshin Kwak, and Nojun Kwak, welcomed me whenever I visited Korea. I truly appreciate our friendship over many years, particularly with Jaehoon Kim.

Lastly, but not least, I thank my family for their love and support. Without them, I would never have finished my program. I am grateful for the support of my brother and sister-in-law, and their two daughters always made me smile. My mother has supported every decision I made throughout my entire life. I deeply appreciate her faith in me. My father taught me how to pursue my goals and not to stray from them. I sincerely wish he could have witnessed me finish my program and receive my Ph.D. degree. I dedicate this dissertation to my father, who was a true scholar in his heart.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Virtualization environments have become basic building blocks in consolidated data centers and cloud computing infrastructures. By running multiple virtual machines (VMs) in a shared physical machine, virtualization achieves high utilization of hardware resources and provides strong isolation between virtual machines. This dissertation discusses the implementation and the evaluation of an approach called kernel service outsourcing that improves the performance and the reliability of guest systems in virtualized, multi-kernel environments. Kernel service outsourcing allows applications to exploit operating system (OS) services from an external kernel existing in the shared system without limiting application OS service requests to the local kernel. Because OS services by external kernels may be more efficient services by local kernels, kernel service outsourcing creates new opportunities for better performance by applications in the guest OS. Moreover, application of the kernel service outsourcing technique implements natural diversity, which improves the reliability of virtualized systems.

We present two major benefits of kernel service outsourcing. First, we show that I/O service outsourcing can improve the I/O performance of guest OSes by up to several multiples. In some important cases, the performance of network applications in the guest OS using network outsourcing was comparable to that of the native OS. We also applied kernel service outsourcing between Windows and Linux, and determined that kernel service outsourcing is viable even with two heterogeneous OS kernels. In addition, we studied more performance optimization techniques that can be successfully implemented in the external kernel when certain OS services are outsourced to the external kernel.

The second benefit of kernel service outsourcing is to improve system reliability through the natural diversity created by the combination of different kinds of OS kernel implementations. Because OS services can be outsourced to different versions or even to heterogeneous types of OS kernel for equivalent functions, malicious attacks that target certain vulnerabilities in specific versions of OS kernels cannot succeed in the outsourced kernels. Our case studies with Windows and Linux show that in the outsourced systems, kernel service outsourcing prevented malicious attacks designed to exploit implementation-dependent vulnerabilities in the OS kernels.

CHAPTER 1

INTRODUCTION

Virtualized environments [5][14][27][31][32][57] have become basic building blocks in consolidated data centers and in cloud computing infrastructures. By running multiple virtual machines (VMs) in a shared physical machine, virtualization achieves high utilization of hardware resources. Live migration and easy restart of VMs improve the manageability of large datacenters. Meanwhile, virtual machine technology provides strong isolation among virtual domains. For example, security isolation prevents a malicious application from attacking applications or accessing data in other domains. Fault isolation prevents one misbehaving application from bringing down the whole system. Environment isolation allows multiple operating systems to run on the same machine, accommodating legacy applications and cutting-edge software, each with a separate set of configurations and parameters.

Typically, guest OSes in the virtualized environments run heterogeneous OS kernels. Some OS kernels have different types of OSes, such as Linux and Windows. Different OS kernels often have different functionalities. For example, some OS kernels have privileges to access hardware devices directly, while others do not. Vulnerabilities found in one OS kernel may not appear in another OS kernel in a different domain. This heterogeneity of OS kernels provides an interesting opportunity for combined kernel service processing. Kernel processing in one domain may be more efficient because of the differences in privileges. Malicious attacks that compromise one guest system may not work in another OS kernel because of the kernel implementation differences.

In this dissertation, we implemented and evaluated a mechanism called kernel service outsourcing. This mechanism allows applications to exploit OS services from an OS kernel of another domain in the physical host (an external kernel). Using kernel service outsourcing, applications in one guest OS can bypass the kernel services of the OS kernel the applications are running on (a local kernel) and forward the kernel service requests to an external kernel best suited for processing the requests.

The concept of kernel service outsourcing among multiple kernels is not entirely new: the microkernel approach [2][12][23][38], which implements high-level OS services through user-level servers, has been researched extensively in the literature. Applications in the microkernel can exploit customized OS services through mechanisms such as inter-process communication (IPC). Researchers have recently revisited the microkernel concept to support better scalability and customized performance in multicore systems. Helios [46] introduced satellite kernels that are customized to heterogeneous cores in the system. Satellite kernels communicate through a uniform set of OS abstractions across the heterogeneous codes. Multikernel [6] suggests a new scalable OS architecture, in which a machine is viewed as a network of independent cores and traditional OS functionalities are moved to distributed processes that communicate through message passing. On the other hand, kernel service outsourcing in virtualized systems allows applications to delegate the processing of kernel services into another kernel.

This dissertation explores two benefits of kernel service outsourcing. First, by delegating the I/O services of a guest OS to a privileged OS or a host OS, kernel service outsourcing bypasses the overhead of slow kernel processing in the guest OS and achieves better I/O performance. This is accomplished by using the efficient kernel processing of a

privileged OS kernel. Our experiments with network and filesystem services in different guest OSes, such as guest Linux and guest Windows, show that kernel service outsourcing improves I/O performance by as much several times compared with a guest OS with para-virtualized devices.

The second benefit of kernel service outsourcing is improved system reliability through the natural diversity created by combining two different kernel implementations. Natural diversity based on kernel service outsourcing has advantages, such as low development cost and application backward compatibility, and provides an effective defense against malicious attacks targeting implementation-dependent vulnerabilities. We present the effectiveness of a defense based on natural diversity by using three real-world examples of vulnerabilities present in Windows and Linux kernels.

1.1. Contributions

The first contribution of this dissertation is its demonstration of the feasibility of kernel service outsourcing with real-world applications and showcasing of effective usage cases that benefited from our approach. To show that the proposed approach is a feasible solution in practical environments, we applied kernel service outsourcing to several different guest OSes, such as Linux and Windows, in order to outsource such kernel service processing as network and filesystem of the guest OS to the host OS kernel. In our experiments, network service outsourcing significantly increases the throughput of network applications, matching the performance of native systems in some important cases. We also applied the network service outsourcing mechanism to fully virtualized guest systems and delegated network processing services between two heterogeneous operating systems, Windows and Linux. In this study, we show that the outsourcing

mechanism is a viable solution even with two heterogeneous OSes by enhancing the fully-virtualized Windows network performance by several times.

The second contribution of this dissertation is its detailed analysis of the impact of kernel service outsourcing on system performance and resource usage. Using low-level system characteristic measurement tools, such as dstat and kvm_stat, and a profiling tool, OProfile, we monitored low-level system characteristics, such as CPU utilization, the number of total instructions executed, number of context switches, number of emulated privileged instructions, L2 cache misses, and TLB misses. Our measurement results reveal that kernel service outsourcing for network and filesystem significantly reduces such system resource usage as CPU utilization.

The third contribution is to present a collection of techniques that improve system performance in guest operating systems. Techniques that communicate via shared memory and reduce kernel-user level boundary crossings can significantly increases system performance. We present our case study with user-level kernel service and demonstrate how a collection of these techniques provides better network performance in a guest OS. Furthermore, application of these techniques in the external kernels can lead to several multiples of better performance by applications running in the guest systems for such activities as file-socket direct data transfer.

The fourth contribution of this dissertation is to introduce a way to improve the reliability of a system. Kernel service outsourcing can be used to implement the natural diversity of OS services by combining different kernel service implementations. To demonstrate the practicality of our approach, we present a few instances of real-world vulnerabilities in widely used OS kernels (Windows and Linux) and show how kernel

service outsourcing effectively prevents malicious attacks from succeeding in exploiting these vulnerabilities.

The final contribution of this dissertation is to present a methodical way of implementing kernel service outsourcing in various operating systems and virtualization platforms. The great variety of operating systems and virtualization platforms prevents kernel service outsourcing from being easily deployed in different environments. We present a methodical, step-by-step approach for kernel service outsourcing and show how this method can be applied to implement network and filesystem service outsourcing.

1.2. Organization

The remainder of this dissertation is organized as follows. Chapter 2 is devoted to background and related work. In Chapter 3, we describe kernel service outsourcing and evaluate the improvement in performance that kernel service outsourcing brings. In Chapter 4, we present the mechanism that increases system reliability through the natural diversity based on kernel service outsourcing. We describe in Chapter 5 our methodical approach to the application of kernel service outsourcing to virtualized systems. And in Chapter 6 we conclude the dissertation with a discussion of future directions for research.

CHAPTER 2

BACKGROUND AND RELATED WORK

Virtual machine (VM) techniques have been explored in earlier efforts to share expensive mainframe hardware by many users of different requirements. One of the earliest was IBM VM/370, which used virtualization to support legacy binaries [27]. More recently, virtualization techniques have become widely used in data centers to implement server consolidation that improves resource utilization yet provides the isolation between virtual machines.

Despite advantages virtualization provides, virtualized systems often suffer from poor I/O performance due to the overheads incurred by the virtualization layer. Many recent research works have focused on improving I/O performance of virtualized systems. The para-virtualization techniques [5][14][64] improve the guest OS performance by modifying the guest OS kernel code and applying new virtual devices such as the split virtual device driver model [5]. Other researchers were able to improve guest I/O performance by VMM-bypassing [40]. However, the I/O performance problems still exist for some important cases, for example, the I/O performance of hardware-assisted full virtual machines (HVM) still remains low because the limitation of being unable to modify the guest OS kernel code.

Bochs project [70] emulates a number of different x86 processor environments on commodity OSes. UMLinux [13][31] and UML [18] are direct OS port user-level operating systems. King et al. [31] described the overhead associated with UMLinux;

frequent host context switches, protecting kernel space, and switching between applications. They reduced frequent host context switches by moving some of the UMLinux functionalities into the host kernel. Aggregated system calls, which we are introducing in Chapter 3, follow a similar approach. The other two sources of overhead described [31] were removed by SKAS host patch in recent releases of UML.

Researchers made efforts to improve the network performance of user-level VMMs by context switch reduction, copy avoidance and network stack specialization [31][34]. Although their approaches focus on optimization and specialization of guest OSes, the kernel service outsourcing approach improves network performance by bypassing the entire processing in guest OSes.

PlanetlabOS [7] enables a number of users to share the same hardware, providing a virtualized user-level working environment to each user. However, PlanetlabOS, implemented by using Linux vserver [77] and SILK [8], is not a fully virtualized OS because its network subsystem is not virtualized.

Many research efforts have focused on achieving efficient packet processing through collapsing layers. Integrated layer processing (ILP) [1][17] increases performance by reducing redundant copying and buffering. Synthesis kernel [50] collapses layers by in-line code substitution and applying factoring invariants for further optimizations. In this dissertation, we apply code specialization techniques to the network stack code of user-level operating system for improved network performance.

In addition to ILP, several schemes have been proposed to move data between layers without copying. Chu [16] describes a zero-copy TCP protocol stack implementation for Solaris using page remapping and copy-on-write. Fbufs [20] uses

shared memory space to move data between different address space domains. Our shared socket buffer technique uses the same idea of fbuf.

Other research efforts have been put to improve network performance of guest OSes by optimizing device drivers. In the article [43], researchers introduced a new virtual network device driver that incorporates common hardware optimization techniques such as TCP/IP checksum offloading for better guest OS network performance. VMware Workstation batches emulation of several I/O instructions to reduce the number of context switches [61]. VMware Virtual Machine Interface (VMI) provides paravirtual I/O functions, with which a guest OS can use paravirtual network drivers. While these approaches focus on device-level modules based on paravirtualization, our kernel service outsourcing approach provides virtualization at the API level.

XWay [30], XenLoop [63], and XenSocket [67] accelerate inter-VM communication in Xen by using shared memory and other communication support of Xen. These approaches focus only on improving inter-VM communication performance. Our network service outsourcing technique was able to improve inter-VM communication performance by eliminating redundant network processing in the guest OSes. We further improved inter-VM network performance by applying a specialization technique that exploits shared memory between VMs. We present our measurement results in Chapter 3.

TCP/IP offloading [24][29][55] delegates certain network processing to hardware. Other researchers [54] used dedicated CPUs in symmetric multiprocessors (SMPs) and dedicated cluster nodes for delegating network processing. Applications must communicate the dedicated CPUs and nodes with shared memory and System Area Network (SNA). Kumar et al. [36] accelerated the communication processing using

network processors. Raj et al. [51] presented the notion of self-virtualized I/O by offering virtual interfaces, through which guest domains can access physical devices, improving the network throughput and reducing the latency in their experiments using IXP24-based boards. Some researchers [26] used network processors to increase the computation/communication overlap in the system by dynamically mapping different stream processing to best-suited platforms. The kernel service outsourcing for network processing can be viewed as software implementation of techniques such as TCP/IP offloading. However, unlike other approaches, our kernel service outsourcing approach does not require special hardware or dedicated system resources.

In Menon et al. [43], 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 [58]. The VMware Virtual Machine Interface (VMI) provides I/O facilities for paravirtual network drivers in the VMware Workstation [61]. These approaches focus on low-level modules based on paravirtualization while we focused on a high-level module based on kernel service outsourcing.

Researchers have also put their efforts on virtual disk. Ventana [47] introduces virtualization-aware distributed filesystems, offering the benefits of powerful versioning, security, and mobility properties of virtual disks. The O2S2 architecture [52] provides object-based storage device services to virtual machines, enabling efficient sharing of physical devices, dynamic access control, and usability-based performance isolation in heterogeneous storage environments. User Mode Linux [18] includes a special file system called hostfs to access the files in the host OS. Cooperative Linux (coLinux) [3] is a port

of Linux to Windows as well as Linux. Cooperative Linux includes a special file system called cofs to access the Windows files. The hostfs filesystem in UML follows the similar approach as the kernel service outsourcing for the filesystem. However, our kernel service outsourcing achieves better performance by using efficient techniques such as DLL injection to intercept systems calls from applications, while UML suffers from heavy overhead of intercepting system calls [34].

In addition, the virtualization of Graphics Processing Units (GPUs) has also been explored. Gupta et al. [28] presented the GViM system designed to virtualize and manage the resources of a general purpose system accelerated by GPUs.

Diversity is an important natural defense technique to increase the survivability of species in an epidemic outbreak, e.g., during the spread of a novel virus. Researchers have applied diversity concepts to computer software in an attempt to improve the reliability of the software. N-version programming [4] increases software reliability by implementing different versions of software components for the same specification. However, n-version programming typically requires high implementation and maintenance costs.

With the evolution of information technology, recent applications of diversity techniques have been more encouraging. Automated techniques, sometimes called artificial diversity [65], have been successfully demonstrated to provide effective defenses against specific classes of software viruses. A concrete example is address space layout randomization (ASLR) [68], a technique that prevents attackers from predicting the specific location of codes and data by diversifying memory layout of software.

In contrast to artificial diversity, kernel service outsourcing explores the use of natural diversity among different currently existing operating systems (OS) such as Linux

and Windows, to defeat attacks intended for one system but will not work on the others. Compared to artificial diversity, natural diversity has three major advantages. The first advantage of natural diversity is its effectiveness in defeating attacks that exploit vulnerabilities specific to an OS. The effective defense relies on the wide range of differences in kernel interfaces and implementations of naturally diverse OSes such as Windows and Linux. These differences prevent an attack from working simultaneously on two such naturally diverse OSes. The resulting combination achieves the software reliability advantages originally expected of N-version programming: independent failure modes among the versions.

CHAPTER 3

ENHANCING SYSTEM PERFORMANCE THROUGH KERNEL SERVICE OUTSOURCING

3.1. A Performance Study of User-level Kernel Service

3.1.1. Problem Statement

User-level Operating Systems (ULOSes) are operating systems that run "over" other operating system kernels as user processes. Many type-II virtual machines follow the ULOS approach. As a user process, a ULOS redefines its own core functionalities by using host OS interfaces such as system calls instead of the instruction set of the underlying processor. A ULOS provides a set of qualities that enables server consolidation.

Resource allocation. Each guest operating system is a user-level process. Resources such as CPU and memory are allocated according to the host OS' sharing and scheduling policy. Idle resources are allocated to busy processes to increase utilization, while maintaining fair sharing.

Easy maintenance. A virtual network server on a ULOS is easily migrated, paused, and recovered using traditional process migration and recovery techniques. System administrators can easily expand system capacity by adding more hardware and migrating the virtual servers to the new hardware.

Strong isolation and reliability. Since a ULOS is indeed a user-level process, isolation among guest ULOSes is naturally achieved by a host OS's process encapsulation.

When a guest operating system is compromised, the fault is sandboxed and can not affect the host OS nor other guest ULOSes.

Easy installation. In contrast to type-I VMM approaches that need installation of VMMs on bare hardware, ULOS approaches install guest ULOSes on a host OS. This reduces hardware issues in installation, such as incompatibilities with the underlying hardware configuration.

Easy system diagnosis. For diagnosing a ULOS, system administrators are able to use common tools, such as gdb and oprofile, installed in a host OS without requiring special VMM support or kernel patches.

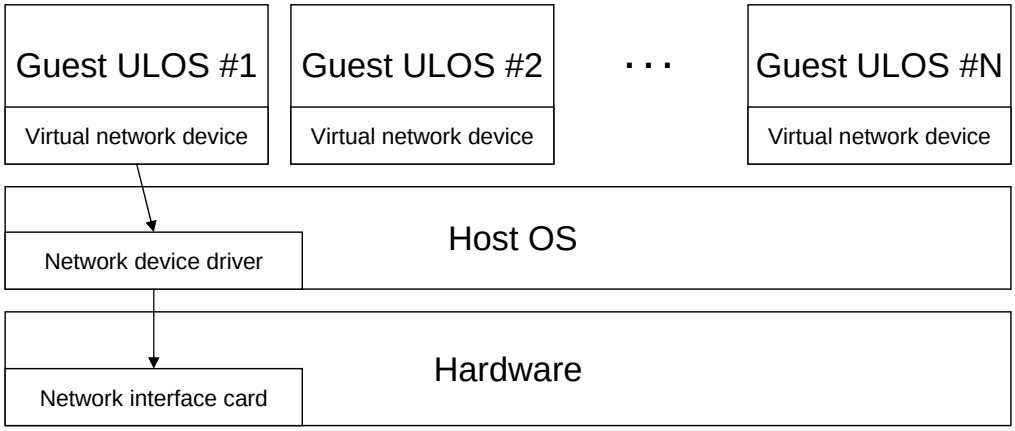


Figure 1 User-level operating system architecture

On the negative side, ULOSes suffer from significant performance penalties for obvious reasons: running at user level, they must invoke the underlying host OS kernel to provide kernel services that cannot be emulated. While this indirection provides strong isolation, it introduces overhead considered to be unavoidable. From the server-

consolidation point of view, it significantly increases the packet-processing overhead, reducing the maximum network throughput and increasing packet latency.

In this study, we have chosen one of the direct OS port ULOSes, User-Mode Linux (UML) for our experimental ULOS. UML is a port of the Linux kernel that runs as a user process on native Linux. It supports the full Linux API through the UML core.[1] When privileged kernel functions are invoked, the UML core calls the host Linux kernel to actually carry out these functions. The support for I/O devices such as network devices is provided through corresponding virtual devices.

We measured the network performance of UML+Linux over a gigabit network. In our preliminary experiments, UML+Linux exhibited considerably poor throughput and latency characteristics. Figure 2 compares the throughput of UML+Linux and native Linux, showing that Linux outperforms UML+Linux by 1.5 to 3 times. We also observed a 10 fold increase in packet processing time in UML+Linux.

[1] In this dissertation, the term “UML core” refers to the OS code that normally would run in kernel mode. Although the term “UML kernel” is the normal usage in the community, the slightly different term “UML core” avoids the overloading of the word kernel.

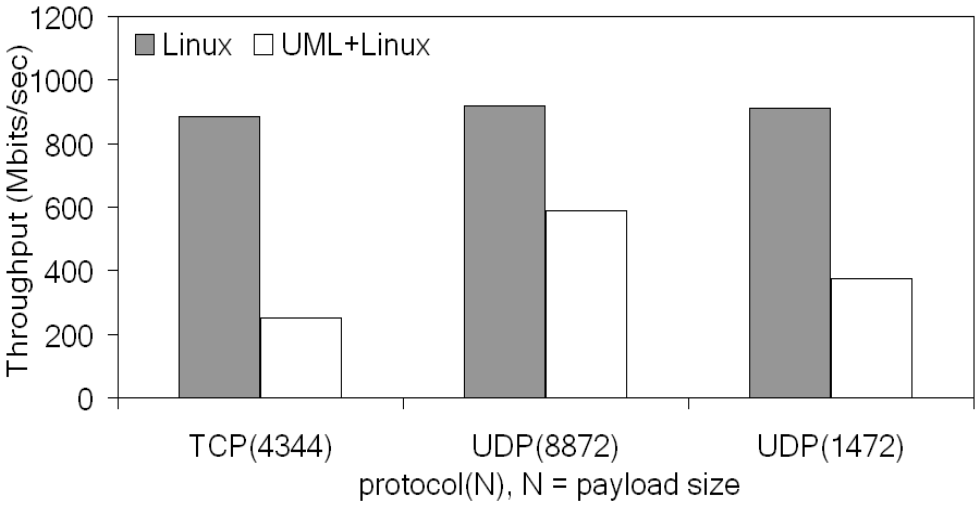


Figure 2 Network throughput in Linux and UML+Linux

We analyze the overhead of a ULOS, dividing the sources of overhead into three main categories:

- Privilege management: a ULOS executes privileged instructions and provides kernel-level services by invoking the host kernel.
- Memory management: Extra memory copies to move data through the ULOS core and additional virtual address translations.
- Additional software instructions: More instructions to be executed due to the ULOS layer. (e.g., virtual I/O devices)

3.1.1.1. Privilege Management Issues

A ULOS reuses the large majority of the kernel code of an existing OS. However, this simple reuse of the existing code raises an impedance mismatch between the code originally written as kernel code and its execution environment in user mode. In particular,

the design of the kernel code depends on low-impedance base operations, assuming that executing privileged instructions and accessing kernel data structures are simple and cheap. This assumption does not hold in a ULOS. The increased cost of base operations entails a high-impedance design for the ULOS.

An important factor leading to the high impedance of base operations in ULOS is frequent and expensive user/kernel boundary crossings that typical ULOS facilities (e.g., disabling interrupts) trigger to implement privileged operations. Boundary crossings are expensive; with a Pentium4 processor machine used in our experiments, the `getpid()` null system call requires more than 1000 cycles (around 0.37 μs) to complete.

3.1.1.2. Memory Management Issues

Another major source of overhead in a ULOS is extra data copies between the added layers. Since the ULOS core inserts a layer between an application and the host kernel, a network packet from the application is copied twice for below layers, the ULOS core and the host kernel. Note that native Linux requires only one copy between the application and the kernel. Our measurement with MTU-sized UDP packets shows that the packet payload copy accounts for around 40% of the latency measured in the virtual network device.

The next overhead related to memory management is introduced by virtual address translation. While the virtual-to-physical address translation in a host OS leverages on fast hardware such as the translation look-ahead buffer (TLB) and hardware-supported page-table manipulation, the address translation in a ULOS is implemented entirely in software. This software implementation naturally suffers from the additional memory accesses for traversing page tables and inefficient error handling without hardware support.

3.1.1.3. Additional Software Instructions

Since a ULOS adds an extra layer between applications and the host operating system, packet processing in a ULOS consumes more instructions. First, packets cross more protection boundaries and consume instructions at each crossing. Second, each packet goes through both the ULOS core and the host kernel, potentially causing extra context switches.

One way to reduce the extra overhead from the layered architecture is to specialize the code for a given context [41][50]. Particularly because the network packet processing has the tendency to have static parameters, techniques such as program specialization can help reduce the overhead [10][11].

To illustrate the impact of the overhead on packet processing, we divide the network protocol stack into five layers. For concreteness, we use outgoing UDP packets in this analysis.

1) ***The user/UML core boundary crossing.*** An application invokes a `sendto()` system call. Control is transferred to the UML core.

2) ***Packet processing in the UML core.*** The UML core executes the usual steps in packet processing, including routing decisions, header filling, network queue processing, and packet forwarding.

3) ***The virtual network device.*** The UML core sends the packet to the virtual network device, which passes the packet to the host kernel. Control is transferred from the UML core to the host kernel.

4) ***Packet processing in the host kernel.*** The host kernel forwards the packets to an appropriate physical network device, e.g., an Ethernet bridge.

5) ***The physical network device.*** Finally, the physical network device driver sends out the network packets through the Network Interface Card (NIC).

We measured the time spent in each layer. The execution time at each layer is divided into two parts: (1) before invoking the lower layer, and (2) after returning from the lower layer. Figure 3 shows our experimental results for MTU-sized[2] packets in Linux and UML+Linux. Note that significant additional latency is introduced by UML in the top three layers.

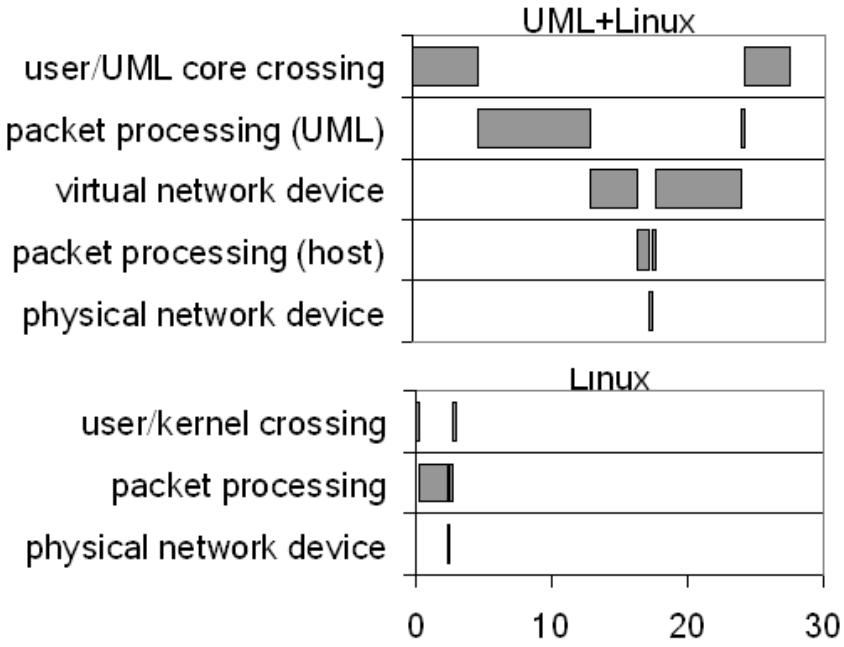


Figure 3 Per-layer latency of UML+Linux and native Linux. The x-axis represents latency in μseconds. We use MTU-sized UDP packets for latency analysis.

[2] The maximum transmission unit (MTU) size in this dissertation is 1500 bytes, unless specified otherwise.

3.1.2. Optimized User-level Network Kernel Service

Despite the non-trivial overhead outlined in the previous section, we were able to alleviate those problems through a combination of system optimization techniques. First, we solve privilege management issues by reducing the need for frequent user/kernel boundary crossings and aggregated system calls. Second, we introduce an address translation cache and shared socket buffers to resolve memory management issues. Third, we apply program specialization techniques to collapse software layers in the ULOS network stack, decreasing the total number of instructions executed. We have implemented these techniques in an experimental ULOS called Enhanced User-mode Linux (EUL).

In the following paragraphs, we list concrete performance problems and the techniques applied to solve those problems.

3.1.2.1. User-Level Signal Masking (ULSM)

Problem: UML Signal Overhead. Disabling interrupts is a cheap synchronization mechanism in uniprocessors to share kernel data structures. Many critical sections in Linux are protected by `cli/sti` assembly instructions along with a few stack operations for saving/restoring the current interrupt flags.

Meanwhile, UML handles interrupts using process signals from virtual devices to the UML core (shown in Figure 4). Therefore, the UML core disables virtual interrupts by masking the signals using the `sigprocmask()`system call. Saving the interrupt state for nested critical sections is implemented by the same `sigprocmask()`, which returns the previous value of the signal mask. Consequently, disabling interrupts, cheap in the Linux kernel, becomes expensive in the UML core, because invoking a system call is costly.

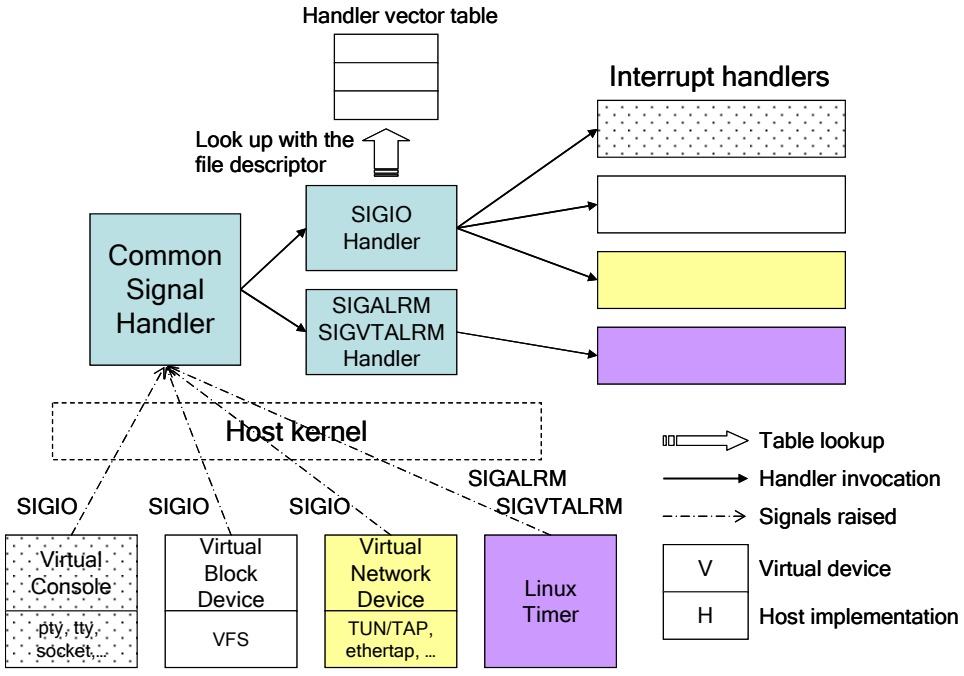


Figure 4 Interrupt handling in UML+Linux. Virtual devices raise SIGIO or SIGALRM signals for new events. The common signal handler in the UML core receives the signals and invokes appropriate handlers in turn.

Solution: User-Level Signal Masking. To avoid using `sigprocmask()`, EUL implements user-level signal masking. EUL removes the host system call by keeping the signal states in the user level (i.e., in the EUL core) rather than in the host Linux kernel. By toggling the interrupt state and keeping track of pending interrupts, the EUL core achieves the same synchronization without the host kernel intervention.

For evaluation of user-level signal masking, we measured the number of system calls invoked by the UML and EUL core for a `sendto()` system call that sends a UDP packet. We show the results in Table 1.

Table 1 Latency of a sendto() with a UDP packet. The payload size of 1472 bytes fills one full MTU.

Payload size	1472 bytes	
Operating systems	UML+Linux	EUL+Linux
# of sigprocmask()	20	0
Time (μs)	27.83	16.37
95% Confidence Interval (C.I.)	0.34	0.30

For each packet, a `sendto()` in UML+Linux requires 20 `sigprocmask()` host system calls: four pairs of interrupt disabling/enabling in the protocol stack and one in a virtual device driver. EUL+Linux requires no system calls, resulting in 42% less elapsed time. Figure 5 illustrates the reduced overhead at the packet processing and virtual device layers.

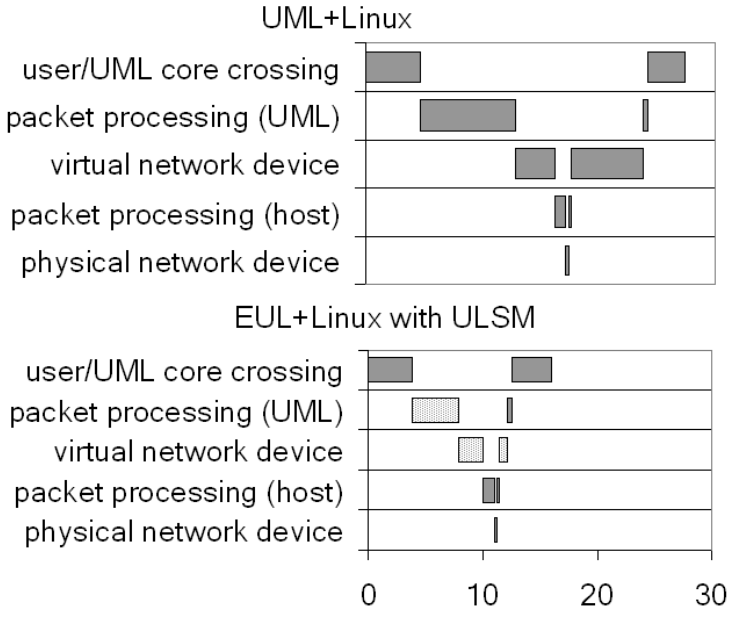


Figure 5 Latency gains due to user-level signal masking

After we started the implementation of the EUL core (based on the UML core version 2.4.26), the original UML core has also evolved. The UML core version 2.6.12 includes a mechanism called soft interrupts [80], similar to EUL user-level signal masking.

3.1.2.2. Aggregated System Calls (AGSC)

Problem: UML System Call Overhead. The UML core implements UML system calls in five steps. First, the UML core process uses `wait()` for a system call from an application process to be intercepted by the UML core. Second, the UML core uses `ptrace()` to copy system call arguments to UML-core space. Third, the UML core executes the system call for the application. Fourth, the UML core copies the return value to the application space using `ptrace()`. Fifth, the UML core resumes the application by another `ptrace()` and goes to the first step—waiting. These steps add up to three `ptrace()` and one `wait()` host system calls. However, these frequent invocations of those system calls are costly.

Solution: Aggregated Host System Calls. The EUL core avoids the repeated callings of `ptrace()` by expanding the scope of tracing facility slightly. We modified the `wait()` routine in the host kernel. The expanded `wait()` carries out the whole parameter manipulation functions described above. This way, the modified EUL core reduces the user/kernel boundary crossing into one per UML system call, compared with four times in the UML core. In Table 2, we show the improvement of the elapsed time for `getpid()` and `sendto()`.

Table 2 Overhead of ULOS system calls

Syscalls	Getpid()		Sendto()	
Operating systems	UML+Linux	EUL+Linux	UML+Linux	EUL+Linux
Time (μs)	6.83	4.47	16.37	15.08
95% C.I.	0.07	0.05	0.30	0.28

3.1.2.3. Address Translation Cache (ATCA)

Problem: Address Translation Overhead. While virtual-to-physical address translation in Linux leverages on hardware, the UML core implements the address translation in software. This software implementation suffers from the additional memory accesses required for traversing page tables. Also, without hardware support for catching traps, the error handling in UML gets inefficient because a segmentation fault signal must be intercepted by the UML core when an unmapped address is referenced. For the protection from accessing the wrong address, the UML core utilizes `sigsetjmp()` and `longjmp()`. The cost of using `sigsetjmp()` for error protection and walking through page tables has an adverse impact on the network performance. As a concrete example, `sendto()` has five arguments, two of which are the address pointers that cause address translations.

Solution: Address Translation Cache. To speed up the address translation, we added an address translation cache (ATC) to EUL. ATC is a software version of the TLB (Translation Look-ahead Buffer). The prefix of a translated address is stored in a hash table for future reference. This hash table simplifies the virtual-to-physical address translation. In the `sendto()` example, ATC reduces the overhead of two address translations.

Figure 6 shows that ATC reduces the latency for the user/UML core boundary crossing and packet processing layers, where copying the destination address and payload requires address translations.

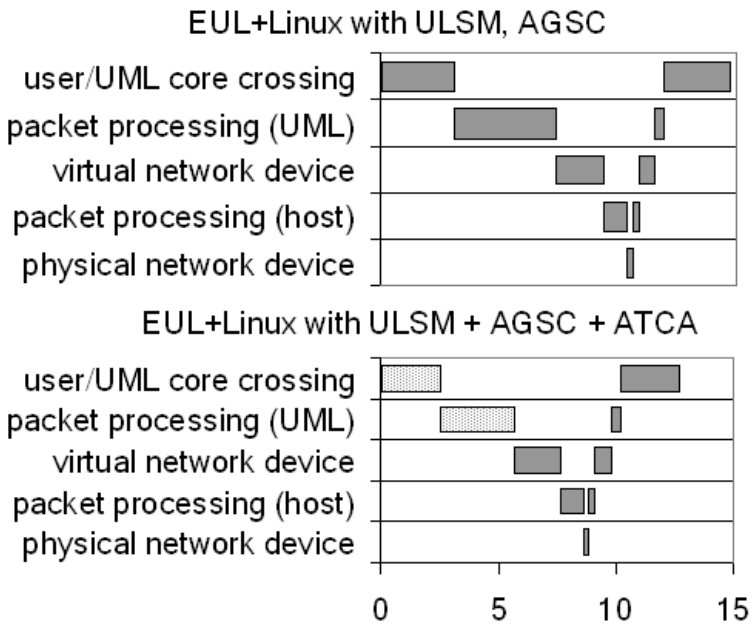


Figure 6 Latency gains due to ATCA

3.1.2.4. Shared Socket Buffers (SSKB)

Problem: Additional Copy across Layers. For applications to send data over network, the Linux kernel copies the packet content once, from the application buffer to kernel space. On the other hand, the UML core copies the packet twice, once from the user buffer to the UML core, then another time from the UML core into the host kernel.

Solution: Shared Socket Buffers. For the implementation of zero-copy between the EUL core and the host kernel, we use a technique similar to fbuf [20]. When an application sends a packet, the packet content is copied into special memory regions shared between the EUL core and the host kernel. The EUL virtual network device passes

the identifier of the shared memory region to the host kernel. ZTAP device (for Zero-copy TUN/TAP) in the host kernel locates the shared memory address from the identifier and creates a socket buffer using the address without copying. Then, the new socket buffer is delivered to a network device.

We measured the packet transfer time spent in a virtual network device of UML and EUL. For the experiments with UDP packets, ZTAP in EUL reduces the elapsed time significantly as shown in Table 3. EUL packet transfer time using ZTAP is about 60% of UML.

Table 3 Elapsed time of a MTU-sized UDP packet transfer in virtual network devices

Virtual network device	TUN/TAP	ZTAP
Elapsed time (μs)	2.90	1.68
95% C.I.	0.06	0.05

3.1.2.5. Network Stack Specialization (NSSP)

Problem: Additional Copy across Layers. Because to the ULOS architecture causes more layers, such as virtual I/O devices, to process network packets, more CPU instructions are to be executed for network processing.

Solution: Network stack specialization. To reduce the number of CPU instructions spent in the multi-layered network protocol stack, we use program specialization [10][11][41]. Program specialization has been acknowledged as a powerful technique for optimizing operating system code for a given execution context. Network protocol stack code particularly provides good opportunities for program specialization [10][11], as network parameters, such as IP addresses and port numbers of peers and socket options, tend to be static once a network connection is established.

The network code specialized for the given context contains fewer instructions and branches by:

- Eliminating the mapping between the file descriptor and the kernel-level socket structure.
- Avoiding the interpretation of socket options
- Avoiding making routing decisions for every `sendto()`
- Inlining layered functions

We use specialization templates generated by the Tempo C specializer [41] to implement specialized `sendto()`[10] in the EUL core. The specialized TCP protocol stack template is filled with the values of IP addresses and port numbers when a TCP connection is established. In the UDP case, we assume that a socket tends to send UDP packets to the same end point. (e.g., in multimedia applications) The template is filled with the process id, the socket file descriptor, and the address of the sock structure. If these values change, the specialized code is invalidated and the EUL core switches back to generic code. The following table shows the gains from network specialization.

Table 4 Specialization impact on UDP processing

Network stack	EUL UDP	Specialized EUL UDP
Elapsed time (μs)	3.23	2.83
95% C.I.	0.04	0.02

To reduce the number of CPU instructions spent in the multi-layered network protocol stack, we use program specialization. The network code specialized for the given context contains fewer instructions and branches by eliminating the mapping between the

file descriptor and the kernel-level socket structure, avoiding the interpretation of socket options, avoiding making routing decisions for every `sendto()`, and inlining layered functions

3.1.3. Performance Evaluation

3.1.3.1. Experimental Setup

We conducted our experiments on machines that have a Pentium4 3.06 GHz processor with a 512 KB L2 cache, 533MHz front-side bus, 1 GB of main memory and a gigabit network adapter card. We used the Linux kernel version 2.4.26 for the host kernel and its corresponding UML core, patched by host and guest modifications from the UML source tree [80].

Our packet processing latency was averaged over 200 runs. We also present the 95% confidence intervals for latency measurements. We use the ttcp tool for measuring the maximum network throughput. Each machine is connected to a gigabit switch.

We show experimental results for four systems: native Linux, UML+Linux, EUL+Linux, and XenLinux+Xen. XenLinux results are added to compare with other virtualization approaches. (XenLinux version 2.6.11 and Xen 2.0.7)

Table 5 shows total packet processing latency for outgoing and incoming MTU-sized UDP packets. EUL+Linux shows less than half the overhead of UML+Linux for both cases.

Table 5 UDP packet processing latency

UDP packets	UML+Linux	EUL+Linux	Reduction
Outgoing	27.47μs	11.85μs	57%
Incoming	40.62μs	18.17μs	55%

3.1.3.2. Sensitivity to Packet Size

Figure 7 shows the elapsed time of the `sendto()` for various packet sizes. Compared with UML+Linux, the latency for small packets in EUL+Linux is lower by about 60%. For large ones that are fragmented, the slope of EUL+Linux curve is less steep than UML+Linux, since EUL has significantly reduced the overhead. For large packets, EUL+Linux incurs only about three folds the overhead of native Linux, compared with about ten folds of UML+Linux.

Figure 8 shows UDP throughput over a gigabit network. Due to the reduced overhead in the EUL core, EUL+Linux outperforms UML+Linux by around three times. For the large-sized packets, the combined optimizations allow EUL+Linux to match the throughput of native Linux even in a gigabit network because the fixed cost per packet is amortized over more bytes.

The elapsed time of `sendto()` for MTU-sized UDP packets is 11.85 microseconds in EUL+Linux. Hence, theoretically, the maximum throughput we can get is 947.7 Mbps, which is larger than the maximum network throughput (around 916 Mbps) of native Linux. For UDP packets with 1024-byte payload, the `sendto()` takes 11.97 microseconds, which limits the maximum throughput to 653 Mbps. The values in Figure 10-3 confirm these calculations.

Figure 9 and Figure 10 show the results of the same optimization techniques applied to TCP protocol stack. We see the same trend as UDP, although the maximum throughput of EUL+Linux remains about 5% less than that of native Linux.

3.1.3.3. HTTP Server Benchmarks

In addition to the micro-benchmarks, we compare the performance of HTTP servers running apache 1.3, using the httperf benchmark. Two httperf clients connected to a gigabit network send requests for 32KB-sized documents to the HTTP server at a constant rate.

Figure 11 shows the throughput of the HTTP server for each setup. The server on UML+Linux can process a maximum of 300 requests/sec, while the one on EUL+Linux 700 requests/sec. (The CPU usage of the machines reaches 100% at the saturation)

Figure 12 shows that the reply time rapidly increases once the server is saturated. Instead of an exponential growth of input queue and response time, the graph shows a long but constant response time at saturation. This is due to a timeout mechanism in httperf clients, which limits the server load. Figure 10-6 also shows that the server on EUL+Linux has a lower response time (by half) compared with the server on UML+Linux during overload.

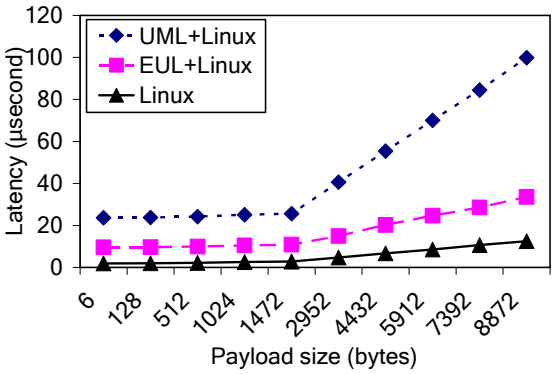


Figure 7 UDP latency with outgoing packets

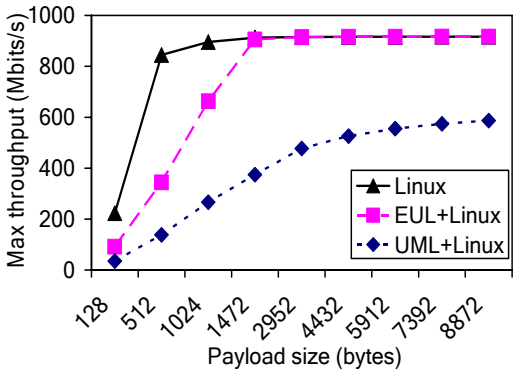


Figure 8 Maximum UDP throughput

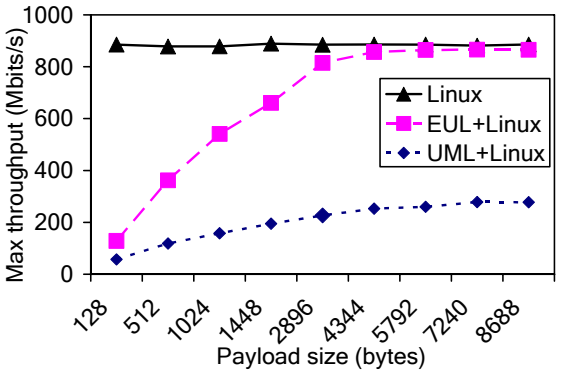


Figure 9 Max. TCP sending throughput

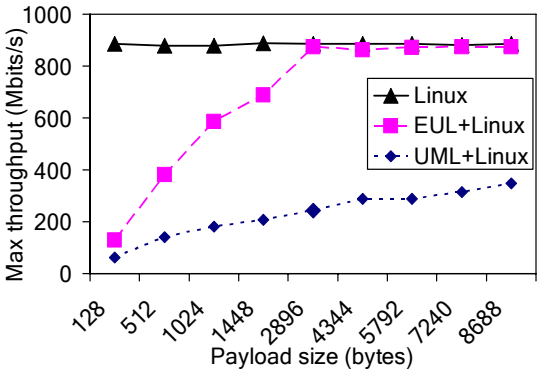


Figure 10 Max. TCP receiving throughput

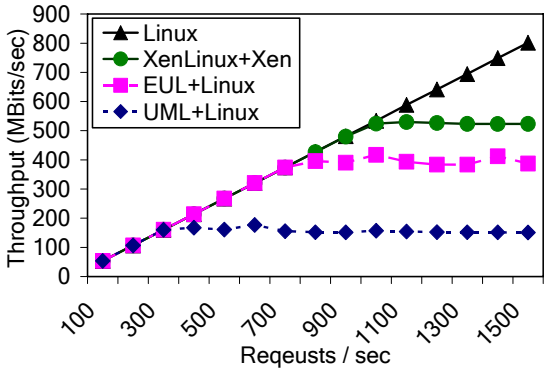


Figure 11 Http server throughput

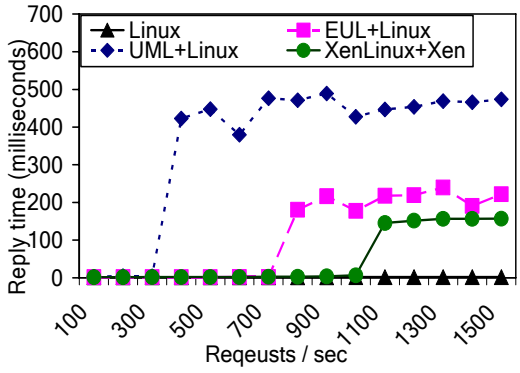


Figure 12 Http server reply time

3.2. Fast Networking through Socket Outsourcing

3.2.1. Problem Statement

Virtual Machine Monitors (VMMs) provide significant advantages in terms of isolation and portability of applications. An early classification [27] of VMMs has divided them into two types: Type I VMMs, which are hypervisor-based VMMs running on bare hardware such as Xen [5] and VMware ESX Server [62], and Type II VMMs (also known as hosted VMMs), such as VMware Workstation [81], Linux KVM [32], and User Mode Linux (UML) [18]. Compared to Type I VMMs, hosted VMMs have advantages such as host operating system (OS) reuse, and OS installation as a normal application program [57], 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 [32] 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 [58]. 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.

3.2.2. The Design of Socket Outsourcing

We present 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-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.

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.

3.2.2.1. Network I/O in VMMs

In this subsection, we illustrate and compare different network I/O mechanisms typically used in virtualized systems.

Figure 13 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 [61].

Figure 14 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. 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.

3.2.2.2. Overview of Socket Outsourcing

While typical para-virtualization devices avoid device emulation, as described in the previous section, by invoking a low-level host driver from the guest OS, socket-outsourcing bypasses the guest OS protocol stack by invoking the socket layer in the host OS,

Figure 15 illustrates the control flow for network I/O processing in socket-outsourcing. While paravirtualization attempts to bypass redundant processing by using a low-level interface (e.g. device drivers), outsourcing bypasses redundant processing by using a high-level interface (e.g. socket API). 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 15, the socket layer is a guest module in outsourcing and it is modified as the device driver is modified in the paravirtualization in Figure 14. 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 15, the host module runs in a user-level process. We can also execute the host module in the kernel.

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.

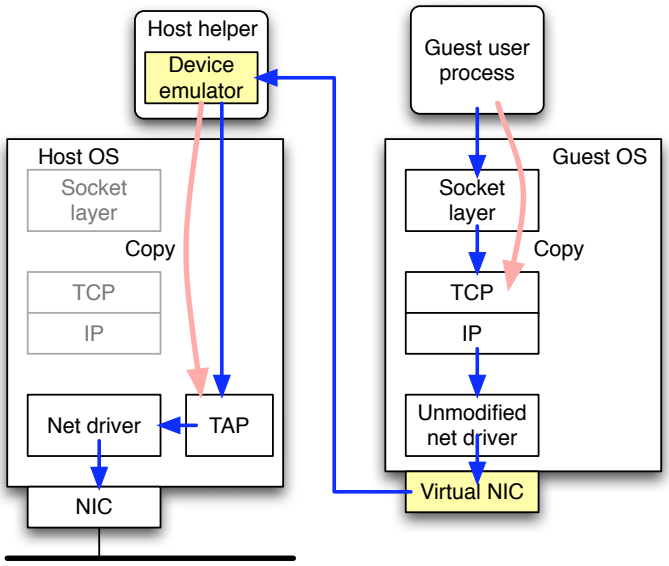


Figure 13 Full virtualization with device emulation

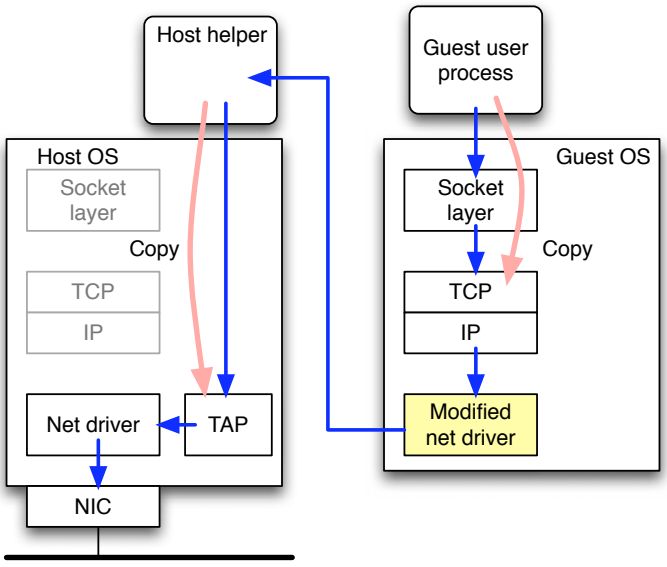


Figure 14 Para-virtualization network model

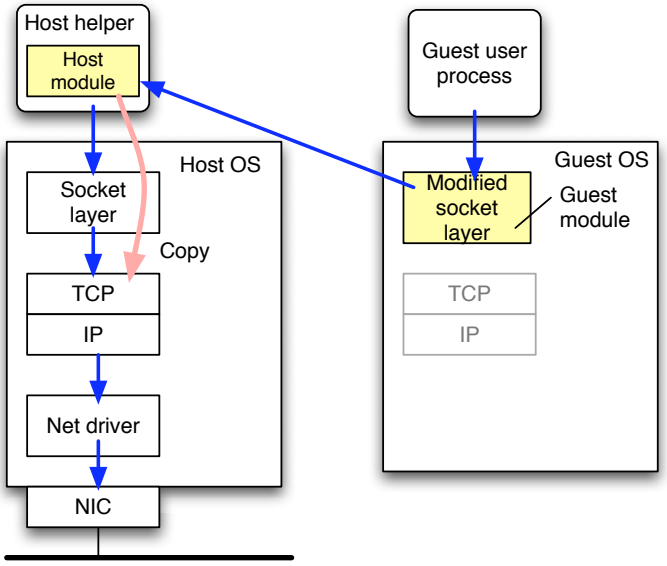


Figure 15 Socket outsourcing network model

3.2.2.3. VMM Support for Socket-outsourcing

To implement socket-outsourcing, the VMM should support communication and synchronization facilities between a guest module and a host module.

Shared memory: For fast communication between a guest module and a host module, 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 queue 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.

VRPC, similar to the remote procedure call (RPC), allows the guest module to invoke procedures in the host kernel. VRPC has the following features for the hosted VMM environment. First, the VRPC server (the host module) does not block. For example, when the guest module invokes the `recv()` call, the VRPC server should return an error immediately with no message 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 design choices simplify the implementation of outsourcing.

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 of the events arrived. Generating interrupts is a common facility of the VMM.

Different events used in the host module to notify the guest module of actions of interest are shown in Table 6. 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.

Using event queues and VRPC, the processing of socket functions in the guest OS is delegated to the host OS. To intercept the socket functions in the guest kernel (Linux), we replaced functions in structure `proto_ops` for TCP and UDP with substitute functions. Among the socket functions in the `proto_ops` structure, we describe how `inet_recvmsg()` is processed with socket-outsourcing to illustrate the key idea behind implementing socket-outsourcing. The `inet_recvmsg()` function is called from not only system call `recvmsg()` but also system calls `recv()`, `recvfrom()`, `read()`, and `readv()` to receive a TCP message.

When an application invokes one of the socket receiving functions and eventually the `inet_recvmsg()` function is invoked, this function first allocates non-pageable memory in the kernel space. Next, it performs a VRPC procedure to the host module. If a message has arrived, the VRPC procedure returns the number of bytes received. In this case, the function copies the message to the user space, frees the non-pageable memory,

and returns the result value. If no message has arrived at the socket, the guest module stops current process and lets the process wait for a new message.

Table 6 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.

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.

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. To allocate one or more dedicate IP addresses to each VM instance, 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.

3.2.3. Performance Evaluataion

3.2.3.1. Experimental Setup

We present our experimental results, comparing with the performance of the baseline case and the socket-outsourcing case. For most of our experiments, 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. 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.

3.2.3.2. Maximum Network Throughput

We used iperf [76], 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 bytes.

We compare the performance of different mechanisms such as device emulation, para-virtualization devices (Virtio), and network outsourcing. Figure 16 presents the

network throughput performance results in the KVM environment. Our experimental results show that the outsourcing mechanism increases the network throughput around by two folds compared with the para-virtualization approach (Virtio) and achieves comparable network throughput to native OSes with larger size of network packets.

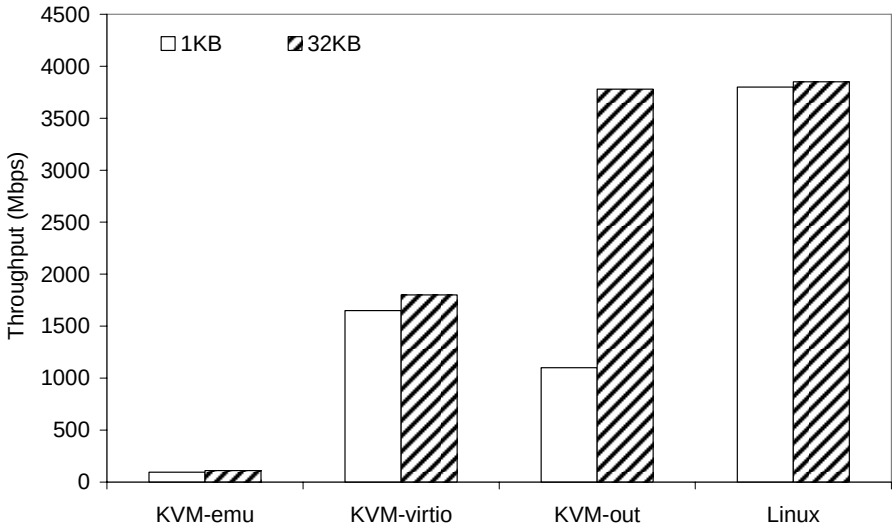


Figure 16 Maximum TCP throughput measured with iperf

3.2.3.3. Web Application Benchmark

To evaluate the impact of performance on applications, we measured throughput with the RUBiS benchmark [15], 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.

Figure 17 and Figure 18 show the RUBiS benchmark throughput measurement results. When we ran both Tomcat and MySQL in a single virtual machine (Figure 17), socket-outsourcing (KVM-out) increased the throughput by 44 percent compared with device emulation (KVM-emu), and showed better performance than that of the para-virtualized Virtio device (KVM-PVdev). Using two virtual machines, in each of which Tomcat and MySQL are deployed respectively, KVM-out improved performance by 46 percent compared with KVM-emu. These results suggest that socket outsourcing has affected the real-world applications and increased the throughput of web services significantly compared with the Virtio case.

3.2.3.4. Inter-VM Communication

We measured the maximum TCP throughput between two virtual machines running on the same host OS by using the iperf tool to measure the throughput of inter-VM communication. For comparison, we also measured the throughput of two processes within a native OS via a local lookback interface.

Table 7 Throughput of inter-VM communication

	KVM-PVdev	KVM-out	Native Linux
Throughput	790 Mbps	5700 Mbps	18500 Mbps

Table 7 present our experimental results. KVM-out was able to reach throughput of 5700 Mbps while the KVM para-virtual device (KVM-PVdev) only achieved 790 Mbps. This performance gain of inter-VM communication comes from socket-outsourcing

eliminating redundant processing of two different network stacks and using only one stack for communicating between VMs.

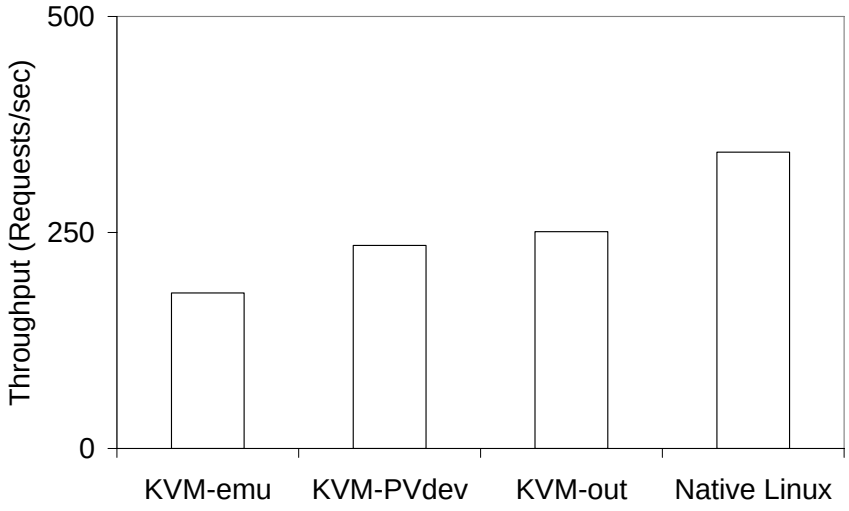


Figure 17 Throughput of RUBiS benchmark in single VM

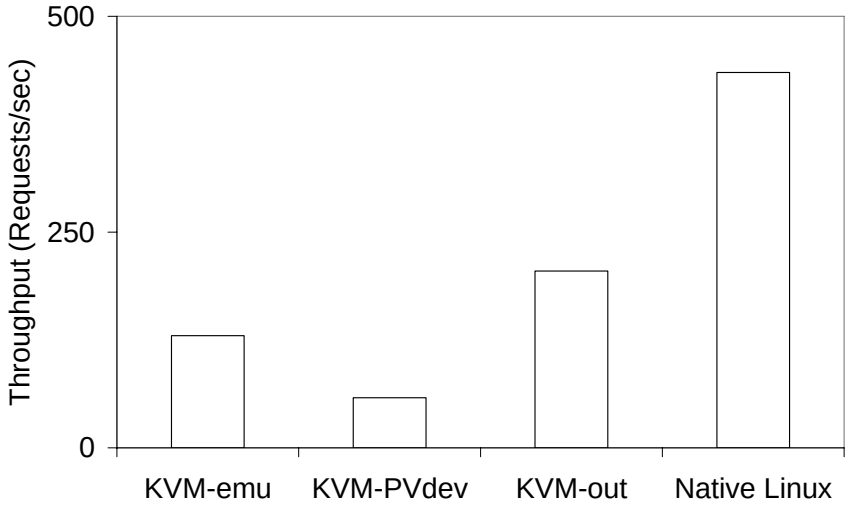


Figure 18 Throughput of RUBiS benchmark in two VMs

3.3. Improving Guest Windows Network Performance (Linsock)

3.3.1. Problem Statement

Full virtualization (FV) has the advantage of compatibility with production operating system (OS) code, but the typical implementation methods (e.g., emulated devices) of FV suffer from significant emulation overhead, particularly I/O processing such as network protocols. Consequently, FV is typically associated with high overhead. To bypass duplicated functions and other sources of overhead in FV, paravirtualization is able to improve the I/O performance through kernel-level system interface modifications in device drivers. However, paravirtualization can only mitigate the virtualization performance overhead partially, typically achieving a fraction of native performance.

Instead of paravirtualization, we apply the outsourcing method [22] to implement FV network processing, achieving significant bandwidth gains compared to device emulation and paravirtualization, even comparable to native Windows in several important cases. Outsourcing bypasses the overhead in the operating system through the adoption of a user-level interface. Concretely, we chose the socket interface (Winsock for Windows and a variation of Linux socket interface) in our implementation. Outsourcing avoids kernel interface modifications and achieves significant performance gains.

In this study, we demonstrate an outsourcing method for full virtualization called Linsock. This method combines Windows as the guest operating system (OS) and Linux as the host OS. A major technical barrier in such heterogeneous combinations is typically the incompatibility of kernel level interfaces between the guest OS and host OS. Consequently, it is difficult to achieve optimization through low-level kernel

programming such as paravirtualization. Outsourcing overcomes this barrier by choosing compatible high-level interfaces, e.g., sockets in Linsock.

We also present a set of experiments that demonstrate Linsock's performance gains in FV. Perhaps contrary to the normal expectations of additional overhead associated with full virtualization, we show the significant performance gains in network protocol processing achievable through network processing delegation in experiments ranging from micro benchmarks to well-known application-level benchmarks. Our experimental results show TCP performance increases of more than 300% compared with device paravirtualization in a 10Gbps Ethernet networking environment. In addition, Linsock also yields a fourfold increase in inter-VM communication performance.

Linsock achieves application transparency without modifying the operating system kernel, but it makes an assumption about network applications running on the guest OS (Windows). Linsock supports network applications that use the standard Winsock API, however other applications that do not use the Winsock API (e.g. InfiniBand applications) will not benefit from the performance gains of Linsock. The application of the outsourcing method to other APIs than Winsock is a subject of future research.

3.3.2. Linsock Approach

To achieve high-speed network performance in fully virtualized guest Windows, we propose an experimental mechanism called Linsock. As in socket-outsourcing in Linux, Linsock delegates network processing of fully virtualized guest Windows to that of the host Linux. Linsock maintains its application transparency by carefully translating Windows socket (Winsock) interfaces to BSD-compatible Linux's socket interfaces.

Unlike Socket-Outsourcing, the Linsock mechanism is implemented mostly in user level, only requiring a simple device driver installed in the guest OS.

The Linsock approach differs from conventional device para-virtualization in a couple of ways. First, although paravirtualized device drivers improve performance by communicating directly with service domain OSes at the device level, Linsock allows applications to facilitate the network processing of host OSes through application programming interface (API). This Linsock design allows applications to bypass slow guest OS's processing and to transfer large network packets to the host OS without segmentation. Second, Linsock requires no modification of the kernel code, neither in the host OS nor in the guest OS. Since most of components are running in user mode, less effort in development and maintenance is required than for paravirtualization techniques, which typically require the modification of kernel-level codes. Additionally, when known vulnerabilities exist in Windows, Linsock provides a way to bypass Windows' network processing without installing patches.

Linsock requires two program modules, a Winsock-Linsock Translator (WLT) and a Linsock server, each of which is running in the guest OS and host OS respectively. The WLT, which runs in the guest Windows, intercepts network applications' Winsock API function calls and transfers data to the Linsock server running in the host Linux. The Linsock server in turn uses host OS network stacks through BSD sockets to process requests for the applications. The Linsock server also notifies the WLT of incoming events, such as packet arrivals. Both the WLT and the Linsock server are running as a user-level process in the guest OS and host OS. Figure 19 illustrates the system architecture of the Linsock approach.

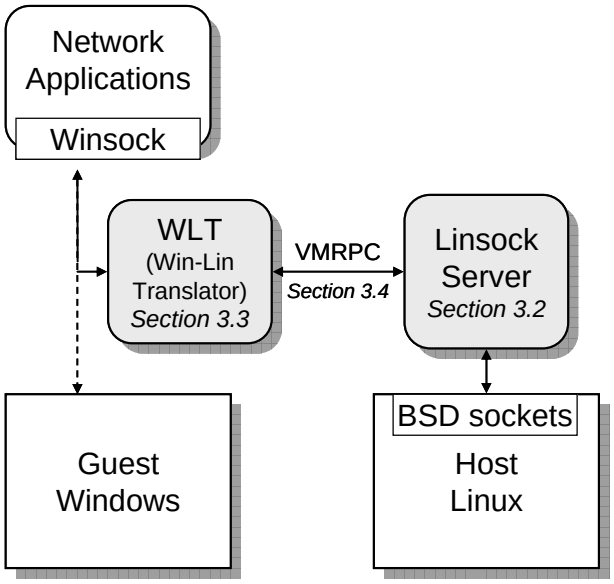


Figure 19 Linsock architecture

When applications invoke a Winsock function, the WLT intercepts and translates the Winsock API function to one or more Linsock API functions, and the WLT remotely invokes the Linsock functions. The Linsock server implements Linsock API functions that utilize the host OS's BSD socket API. Also, the Linsock server monitors network events, such as message arrival, and notifies the WLT of the event. We list Linsock API functions and short descriptions in Table 8. We also define event messages to notify applications of asynchronous events. We list and define in Table 9 the types of Linsock event messages.

Table 8 Linsock API functions

Name	Description
WL_init	Performs initialization. Creates an event queue between WLT and Linsock server
WL_cleanup	Frees up any used resources. Closes open sockets
WL_socket	Creates a socket instance
WL_close	Closes a socket instance
WL_bind	Binds a name to a socket
WL_listen	Listens for incoming connections on a socket
WL_connect	Initiates a socket connection
WL_accept	Accepts a socket connection
WL_sendto	Sends a network message
WL_recvfrom	Receives a network message
WL_shutdown	Shuts down a connection
WL_getsockname	Gets the name of a socket
WL_getpeername	Gets the name of a connected peer
WL_setsockopt	Sets options on a socket
WL_getsockopt	Gets options on a socket
WL_select	Gets the status of a list of sockets

Table 9 Linsock events

Event	Description
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

3.3.2.1. Winsock-Linsock Translator (WLT)

Although Windows supports BSD-compatible socket interfaces such as `connect()` and `accept()`, Windows defines its own socket API, called Winsock [78]. To intercept Winsock functions from applications, we use the Winsock Service Provider Interface (SPI) [78] provided by the Windows OS. The SPI allows service provider software to intercept Winsock functions and implement additional services on top of the base Winsock service. The WLT, which exploits the SPI, is built as a Windows dynamic linked library (DLL). The WLT DLL is automatically loaded to applications that use TCP or UDP network protocols at run time by the Windows kernel.

The Winsock SPI provides 30 different functions for intercepting Winsock APIs. When the WLT DLL is loaded, it registers new function pointers for original Winsock API functions. The following describes some of the SPI functions we intercept. Note that each WSP-* function replaces the corresponding WSA-* Winsock API function.

WSPStartup() replaces the original WSAStartup() Winsock API function for initialization of network processing. Additionally, WSPStartup() allocates memory regions used for shared memory between WLT and the Linsock server.

WSPSocket() creates a file descriptor that will be passed to the calling application. It also creates a Linsock socket by invoking the Linsock API function, `WL_socket()`.

WSPBind(), WSPListen(), WSPAccept(), and WSPConnect() implement the corresponding Winsock API functions, WSABind(), WSAListen(), WSAAccept(), and WSAConnect(), to establish TCP connections. WSPSendto() and WSPRecvfrom() send and receive TCP and UDP messages.

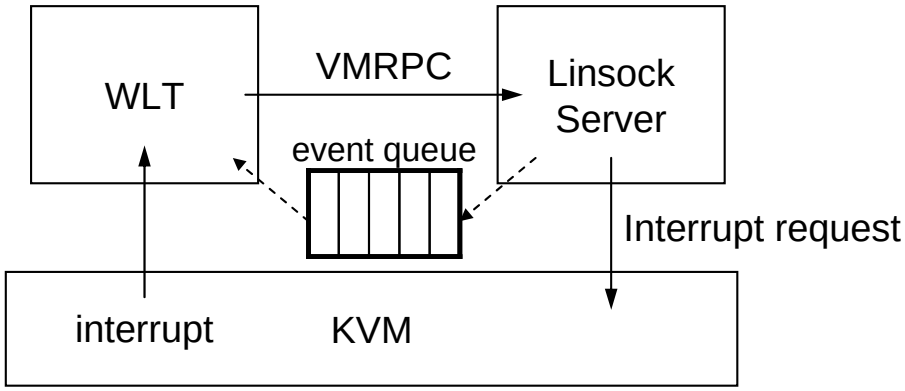


Figure 20 Event queues, VMRPC, and interrupts.

Since the WLT and the Linsock server are running in different domains, one in the guest OS and the other in the host OS, we need efficient and fast communication between the WLT and the Linsock server. Our Linsock implementation provides three mechanisms, event queues, VM Remote Procedures (VMRPC), and VMM-generated interrupts.

Linsock event queues. An event queue, which is implemented as a ring buffer in the shared memory regions, is used for exchanging asynchronous events between the WLT and the Linsock server.

VM remote procedure calls (VMRPC). A remote call invocation from the WLT to the Linsock server is provided by a mechanism called VMRPC. Using VMRPC, the WLT can remotely invoke Linsock API functions and receive a reply. VMRPC is similar to hypercalls in Xen.

Emulated interrupts. When the Linsock server needs to notify the WLT of an event, such as an incoming packet, the Linsock server generates an interrupt to the guest

OS through the VMM. Generating an emulated interrupt to the guest OS is a common function of VMMs that support full virtualization.

3.3.2.2. A Linsock Network Processing Example

We will describe an example of Linsock network processing. Imagine an application that starts TCP/IP networking. The application first calls `WSAStartup()`, which will be intercepted by the WLT through `WSPStartup()`. `WSPStartup()` executes initialization routines such as creating an event queue and invoking `WL_init()` through VMRPC. Next, the application invokes `WSASocket()` to create a socket. The `WSASocket()` system call is intercepted by the WLT, which in turn initiates a VMRPC call for the Linsock function `WL_socket()`. The Linsock server executes `WL_socket()`, issuing the system call `socket()` in the host OS to create an actual socket. The Linsock server starts to monitor the status changes for the new socket. Finally, The new socket handle is returned to the WLT as the return value of `WL_socket()`.

When the application tries to receive data using the socket, the application will invoke the `WSARecv()`, which will be translated by the WLT to the Linsock function `WL_recvfrom()`. If a message has arrived, the VMRPC function returns the number of bytes received. In this case, the `WSARecv()` returns with the received message.

If no message has arrived on the socket, the WLT blocks the current process and waits for a Linsock event message. When the Linsock server notices a message arrival, the Linsock server inserts the Linsock event, `ARRIVED`, into the event queue and asks the VMM to generate an interrupt to the guest OS.

Because the WLT is in user mode, the WLT cannot receive emulated interrupts directly from the Linsock server. In order to deliver the interrupts to user-level processes,

we wrote a simple device driver in Windows. The device driver, called the Linsock event driver (LVDriver), installs an interrupt service routine for a specific interrupt and creates a Windows event object. A user-level program, called the Linsock event dispatcher (LVDispatcher), waits on the Windows event object created by the LVDriver. When Linsock applications start, the WLT automatically registers the applications with the LVDispatcher using Windows inter-process communications (IPC). When Linsock events arrive, the LVDriver signals the Windows event object waited by the LVDispatcher, which in turn notifies the corresponding Linsock applications of the event's arrival.

Once the WLT receives the message arrival event, it unblocks the application process and performs `WL_recvfrom()` to retrieve the received data.

3.3.2.3. Linsock Implementation

We implemented Linsock in the Linux Kernel-based Virtualization (KVM) environment [32]. KVM is a kernel extension (a pseudo device driver) that provides a framework for a VMM, catching execution of privileged instructions and sensitive non-privileged instructions through hardware support (Intel VT or AMD-V). KVM includes a modified QEMU [79] for emulating I/O devices.

Using KVM, a program running in the host OS can access guest logical and physical memory through `ioctl()` on the KVM pseudo device. Therefore, shared memory regions used for event queues can be set up by simply passing the guest address pointer to the Linsock server running in the host OS.

We have extended KVM to provide VMRPC support. (KVM-Linsock) KVM-Linsock redirects the *`vmcall`* instruction generated in the guest OS to the Linsock server. More details of VMRPC implementation are described in Section 3.3.2.6.

Because the host Linux running KVM has native device drivers, network processing in the host kernel is naturally faster than that of the guest OS. Thus, by delegating the guest OS' network processing to that of host OS, we can achieve higher network performance.

3.3.2.4. Winsock-Linsock Translation

The Winsock and BSD-compatible socket API defines different structures and type values for certain functions. For example, `fd_set` structure used in the `select()` call is defined differently in Winsock and in the BSD-compatible API. Also, many of the flag and option values are defined differently. Consequently, the WLT must perform type conversions before it invokes Linsock API functions.

Winsock provides extended functions such as `WSAAcceptEx()`, which is not supported by BSD-compatible socket interfaces. We implement those extension functions by combining several BSD-compatible socket functions.

By design, all Linsock API functions are non-blocking because blocking Linsock functions could stall the entire guest OS. Thus, the Linsock server always replies immediately with the return value. Any required blocking of Winsock API functions is implemented using Window event objects in the WLT. Once the WLT needs to block a function, it waits on a Windows event object, which is signaled when a Linsock event from the Linsock server arrives.

3.3.2.5. Asynchronous Events and Event Queues

Shared memory regions are used to implement event queues between the WLT and the Linsock server. Event queues, which are lock-free ring buffers, are used for

asynchronous communication between the two modules. The implementation of event queues is similar to that of the ring buffer in Xen [5].

Monitoring events on a socket is implemented by exploiting the `epoll()` system calls in the Linsock server. Once the Linsock server notices an event on a socket, the Linsock server first inserts an event message in the event queue and generates an interrupt to the guest OS through KVM.

Asynchronous events are handled quite differently in device emulation, paravirtualized devices, and Linsock. In device emulation, each execution of an I/O instruction causes a trap into the VMM. The cost of handling those traps is expensive. In paravirtualization, data are transferred between a front-end driver and a back-end driver by network frames, the maximum size of which is the MTU of the device. Therefore, the messages larger than the MTU must be divided into smaller network frames. On the contrary, because there is no limit on the message size exchanged between the WLT and the Linsock server, large messages are transferred without segmentation. This Linsock design reduces the number of I/O requests.

3.3.2.6. VM Remote Procedure Calls (VMRPC)

VMRPC is used to execute the Linsock API functions remotely from the WLT. We have implemented VMRPC using the instruction *vmcall* in Intel VT-enabled CPUs. When the WLT initiates a VMRPC call, the WLT first puts the VMRPC parameters in microprocessor registers. Next, it executes the *vmcall* instruction, which causes a trap into the VMM (KVM-Linsock). In Intel's terminology, this process is also known as a VM exit. KVM-Linsock analyzes the reason for the trap and transfers the flow of control to the

Linsock server, which performs the requested function. When the function is completed, the control flows back to the WLT in the reverse direction with the return value.

Because VMRPC parameters are passed via microprocessor registers, no marshaling is needed. VMRPC simplifies the implementation of remote Linsock API invocation with simple interfaces.

3.3.2.7. IP Adresses of Guest OSes

Since guest OSes use network protocol stacks in the host OS, the guest OS and the host OS share the same IP address. To assign different IP addresses to each guest OS, we add these IP addresses to network interfaces in the host OS. When a guest process creates a server socket and calls `bind()`, the Linsock server binds to the corresponding address by changing the arguments of the system call `bind()`. For a client socket, `bind()` is automatically invoked by the Linsock server without an explicit request from the application.

3.3.3. Performance Evaluataion

3.3.3.1. Experimental Setup

We used three Intel Xeon 5160 3.0GHz machines (Dell Power Edge 1900) with 4MB of L2 cache and 2GB of main memory for our experiments. Each machine has two 10Gbps Ethernet network interface cards (NICs), all connected directly to each other without a network switch.

We conducted our experiments using Linux 2.6.25 for the host OS with KVM version 84. The Windows OS we used is Microsoft Windows XP with Service Pack 2 installed. We set the main memory of each guest OS to 384Mbytes. For comparison with

other full virtualization techniques, we also ran our experiments in the Xen environments. We used the Xen hypervisor version 3.2.3.

We compare performance of device emulation and paravirtualized device in KVM and Xen. For network device emulation, we used the e1000 emulation device for our experiments. Virtio [37] and GplPV [83] are open-source paravirtualized device implementation in KVM and Xen, respectively. Table 10 summarizes the implementation used in our experiments.

Table 10 Device emulation and paravirtualized devices in KVM and Xen

VMM	Emulation	PVdevice
KVM	e1000	Virtio
Xen	e1000	GplPV

3.3.3.2. System Call Latency

We measured the latency of several socket system calls to compare the performance of guest system calls with external kernel service calls. In the experiments, we use Linux and Windows for our guest OSes and we outsource the network processing to the host OS, Linux, in KVM. To measure overheads of individual external kernel calls, we created a UDP socket and measured the latency of related calls.

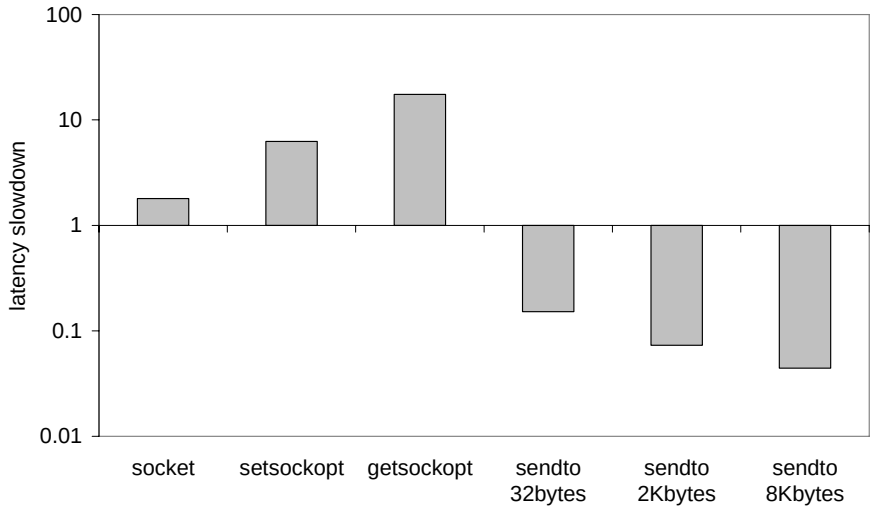
Table 11 shows our experimental results for several socket system calls measured in guest Windows. Calls such as `socket()`, `getsockopt()`, and `setsockopt()` suffered from larger overheads due to external kernel redirection compared with native guest system calls. However, the `sendto()` calls were processed faster from the external kernel, because the `sendto()` calls require significant amount of network processing

and the gain from more efficient processing in the host kernel dominates the redirection cost.

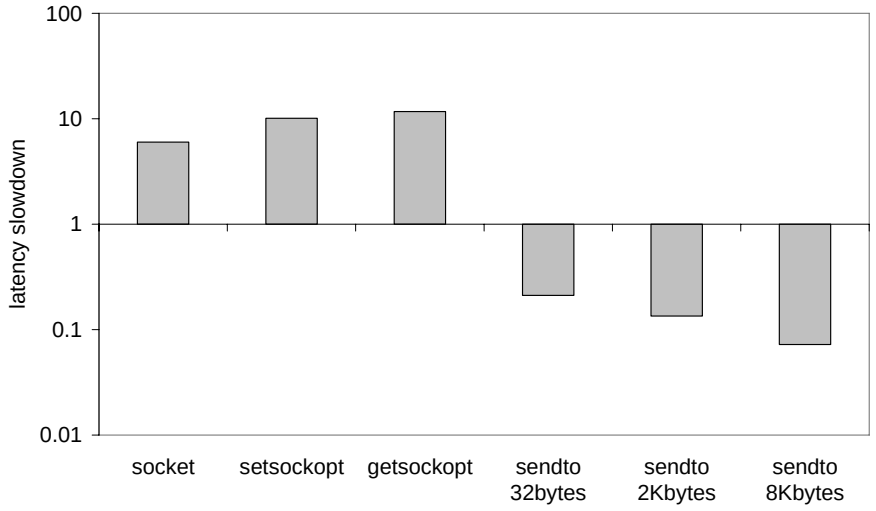
Table 11 Latency comparison of kernel services in guest Windows

	baseline	External kernel service
`socket()`	18.3 us	32.9 us
`setsockopt()`	1.6 us	10.0 us
`getsockopt()`	0.9 us	9.1 us
`sendto()` 32bytes	92 us	14 us
`sendto()` 2Kbyes	233 us	17 us
`sendto()` 8Kbytes	2072 us	59 us

Figure 21-(a) illustrates the slowdown for external kernel service calls compared to the baseline case in a logarithmic scale. The outsourced `sendto()` calls show more speedup with larger UDP packets, which require more kernel processing. The same set of kernel service functions in a guest Linux shows also similar performance implications as shown in Figure 21-(b).



(a) Guest Windows Kernel Services



(b) Guest Linux Kernel Services

Figure 21 Relative performance of kernel services in guest Windows and Linux

3.3.3.3. Maximum Network Throughput

We used iperf [76], a network bandwidth measurement tool, for measuring maximum network throughput between our experimental machines. We changed message sizes from 1K to 64K using iperf option –l for both iperf server and client. For better performance in 10Gbps network, we set the MTU of each NIC (including paravirtualized

NICs) to 9000 bytes. We measured TCP throughput between a guest OS and another physical machine.

Figure 22 and Figure 23 show TCP throughput results in KVM with device emulation, paravirtualized device (PVdevice), Linsock, and native Windows. Device emulation has the lowest performance, saturating its maximum throughput at around 1Gbps, while PVdevices at best only reaches 1.3Gbps. Linsock guest OSes yield TCP throughput of around 5Gbps and 3.1Gbps in sending and receiving, respectively. The TCP sending performance of Linsock guest OSes with larger packets is only 10% lower than that of native Windows. These performance gains mainly come from bypassing the overheads of slow network processing in the guest kernel and avoiding segmentation of large packets. For smaller packets, the overhead of VMRPC becomes relatively higher compared with system calls in the native Windows. Therefore, Linsock suffers from lowered throughput. For TCP receiving throughput presented in Figure 23, Linsock shows around 40% lower throughput compared with native OSes, still twofold better performance than with the PVdevice.

TCP throughput of FV guest OSes in Xen was surprisingly low as shown in Figure 24. The best TCP throughput we were able to reach was 200Mbps when the guest OS was receiving TCP streams with paravirtualized devices.

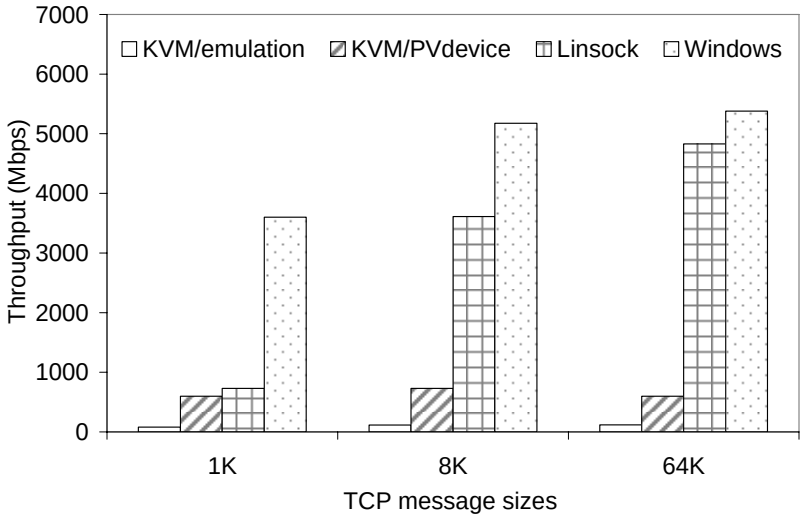


Figure 22 TCP sending throughput

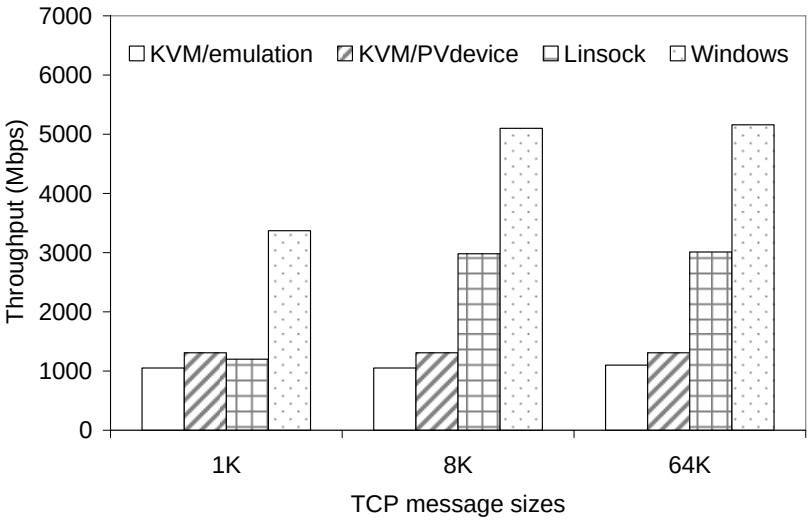


Figure 23 TCP receiving throughput

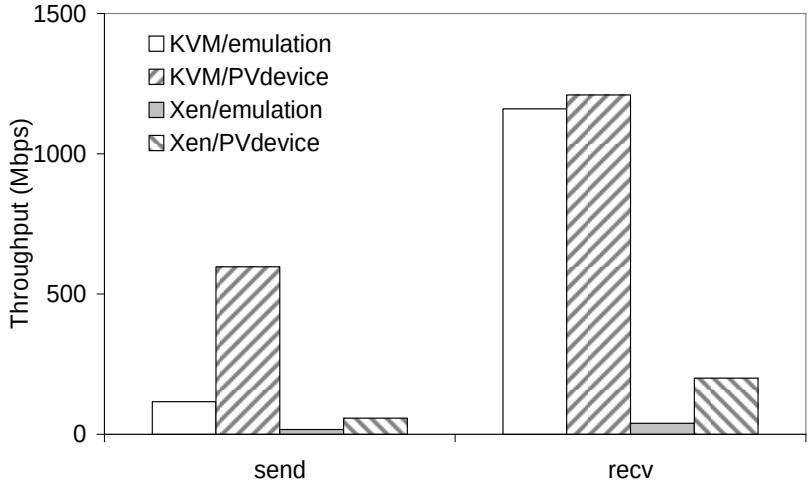


Figure 24 TCP performance in Xen
(message size = 64K)

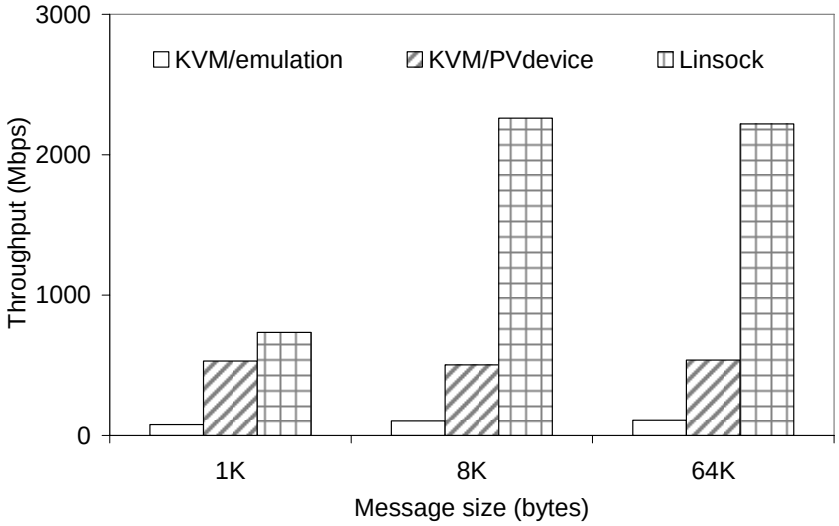


Figure 25 Inter-VM TCP performance

3.3.3.4. Inter-VM Communication

To compare the inter-VM communication performance, we measured maximum TCP throughput between two guest Windows OSes running in the same host OS. Figure 25 presents our results. When device emulation is used, TCP throughput between two VMs is significantly low, saturating at around 100Mbps. Paravirtualization improves performance up to 550Mbps. With smaller messages, the maximum inter-VM throughput with Linsock is around 730Mbps, about a 30% performance increase compared with PVdevice. As message size increases, Linsock's performance increases to 2.2Gbps, a fourfold better performance than other virtualized network I/O mechanisms.

Inter-VM throughput is relatively low compared with TCP throughput over network. We believe that Inter-VM communication requires more CPU resource since both sender and receiver processes are in the same machine. When two machines communicate over network, each peer is responsible for either sending or receiving, which reduces the CPU requirement.

3.3.3.5. Application Benchmarks

To evaluate the performance impact of the Linsock approach on real-world applications, we measure server performance for two applications, file transfer and Web service. For the file transfer test, we chose two different protocols, scp and rsync. We used an Apache Web server [69] and a Web service benchmark, httperf to measure the performance of a Web server.

File transfer. We transferred a 500Mbytes file over a 10Gbps Ethernet network from a FV guest Windows to a client machine running a native OS. Figure 26 shows the elapsed time to transfer the file using two different applications, scp and rsync. As the figure indicates, the Linsock approach reduces the file transfer time significantly, sending the file in 21.3 seconds and 48.5 seconds with scp and rsync, respectively. In contrast, the device emulation and the PVdevice were much slower, requiring more than 100 seconds for the transfer.

Web service. To evaluate Web server performance, we compare network throughput and reply time per http request of an Apache server running in a guest Windows OS. Each httperf client requests 128Kbytes html documents. We set to two seconds the client timeout for http requests. We changed the request rates of httperf clients from 50 requests per second to 500 requests per second.

Figure 27 presents our experimental results for the web server throughput. The web server throughput with device emulation is saturated at around 100Mbps, with the server sustaining up to around 91 requests per second. Other requests were either disconnected because of client timeout (two seconds) or the connection was refused by the system. The PVdevice improves Web server performance, and the Web server was able to

handle up to approximately 400 requests per second. However, more than 400 requests per second overloaded the server and prevented it from handling more requests. The web server throughput decreases as the request rate increases, because more system resources are wastefully spent on handling incoming connections, which will eventually be dropped because of client timeout. On the other hand, the Web server with Linsock was able to handle requests at a rate of 500 requests per second without being overloaded.

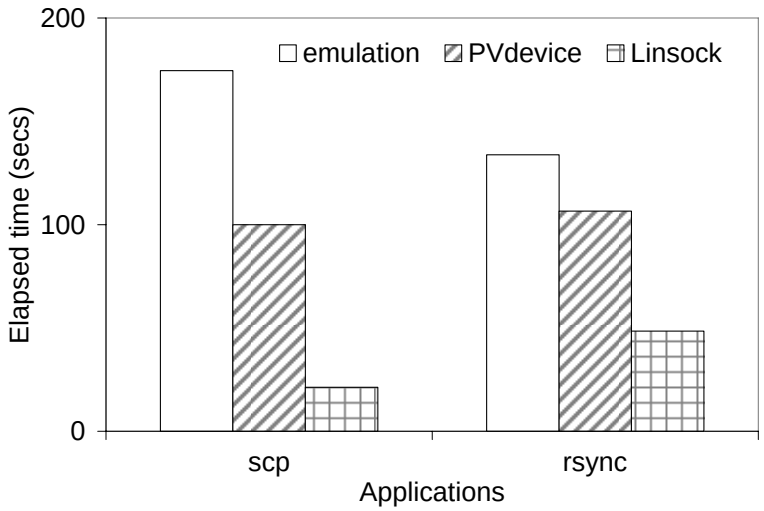


Figure 26 File transfer application performance

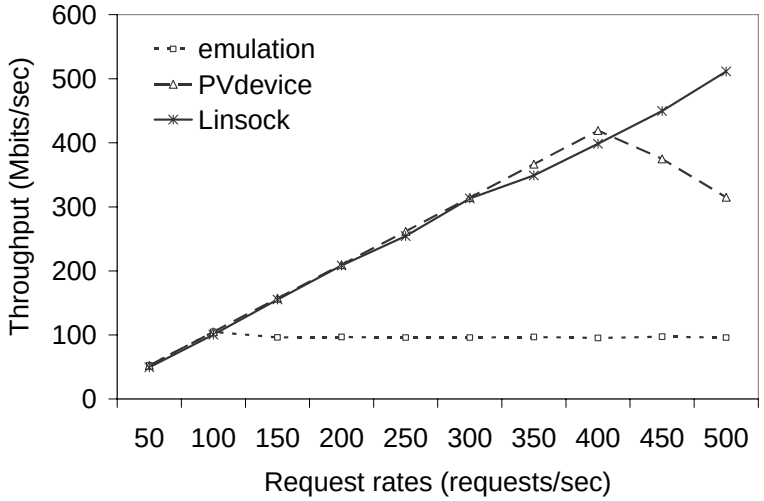


Figure 27 Web server throughput

We also measured the time between when a TCP/IP connection is initiated and when the connection is closed after an http request is completed (reply time). We present, in Figure 28, the measured reply time under a light load. We also show the actual time spent in transferring html documents (transfer time) during the overall reply time. When used with device emulation, the Web server replies an http request in 10.3 milliseconds and in 6.5 milliseconds with PVdevice. The Linsock approach significantly reduces total reply time to 2.1 milliseconds, with 1.6 milliseconds spent for transfer time.

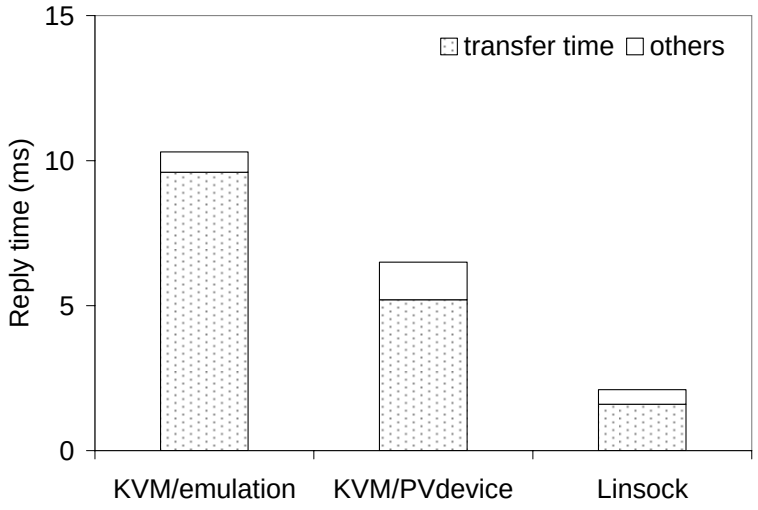


Figure 28 Reply time per http request

3.4. Performance Specialization Through Kernel Service Outsourcing

3.4.1. File-Socket Transfer Specialization

Some socket implementations provide special mechanisms that transfer data read from disks to a socket directly in the kernel. For example, Linux provides the `sendfile()` function, which allows an application to transfer between a file and a socket in the kernel space, eliminating the memory copy overheads between the

application process and the Linux kernel. Windows provides a similar mechanism through the `TransmitFile()` function.

When both file and network services are outsourced to an external kernel, we can further improve this mechanism, eliminating copying disk data into the guest kernel. Avoiding memory copy is particularly important in virtualized environments because of high virtualization overheads involved in copying disk data into/from the guest domain. Figure 29 and Figure 30 illustrate and compare the original `sendfile()` mechanism in the guest OS and the optimization through kernel service outsourcing. We call this special case of kernel service outsourcing, FileTransfer outsourcing.

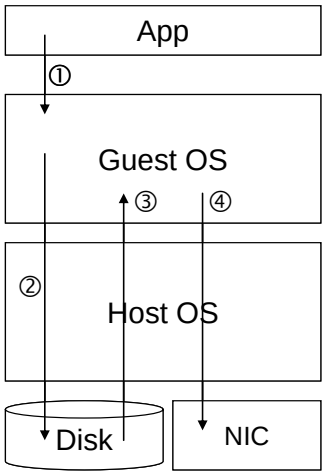


Figure 29 Sendfile() process. Application invokes sendfile() (①) Guest OS reads disk blocks (②, ③) then, Guest OS sends data through network (④)

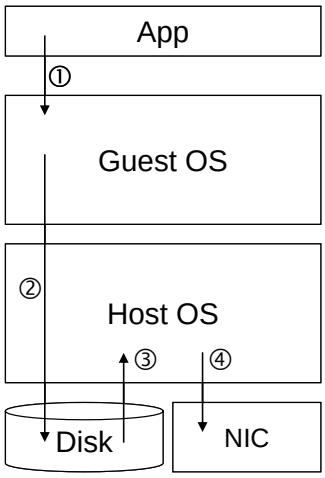


Figure 30 FileTransfer outsourcing. Host OS reads and sends data read from disk (③, ④) without involving guest OS

For performance measurement of FileTransfer outsourcing, we wrote a simple program that sends files over network through `sendfile()` and `TransmitFile()`. In our experiments, we transfer a file, the size of which is 600Mbytes, to a receiver machine. Because the file size is bigger than the memory size of guest OS, we eliminate the buffer cache effect in our experiments.

Figure 31 shows our experimental results. We compare the file transfer throughput of three cases, baseline without kernel service outsourcing, network outsourcing, and FileTransfer outsourcing. In guest Windows, FileTransfer outsourcing improves the file transfer throughput by up to five folds compared to the baseline case and around 50% improvement over network service outsourcing. FileTransfer outsourcing reaches around 400Mbps for its maximum throughput, matching the performance of `sendfile()` in the native Linux. In the guest Linux, the gain from FileTransfer outsourcing is relatively small, still showing around 40% better performance compared with the baseline case, however.

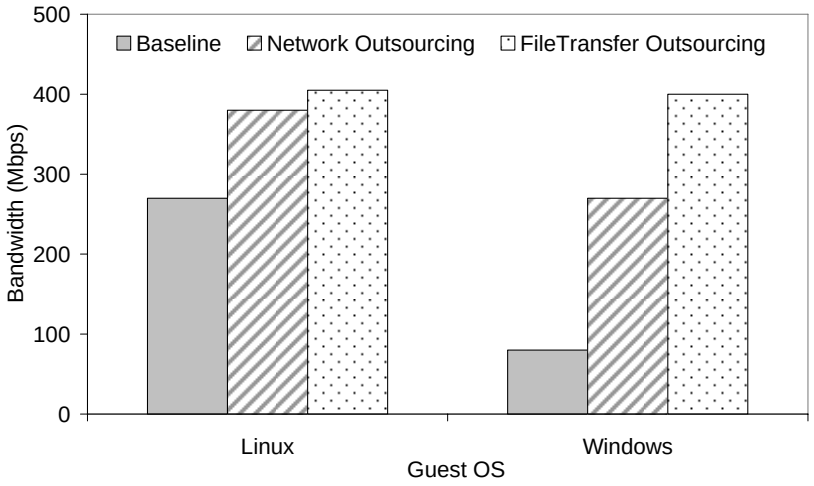


Figure 31 Comparison of Achieved Throughput

We show the performance impact of FileTransfer outsourcing in the real-world applications by measuring file-downloading latency from a web server in guest Windows. For our web server application, we ran the Apache http server [69] with EnableSendfile option on. In the client machines, we executed the `wget` application to download a large file from the web server.

As shown in Figure 32, the latency for file download significantly reduces when FileTransfer outsourcing is applied. When a client downloads a 300Mbyte file, the download latency was only 4.9 seconds, while it was 32.1 seconds without kernel service outsourcing.

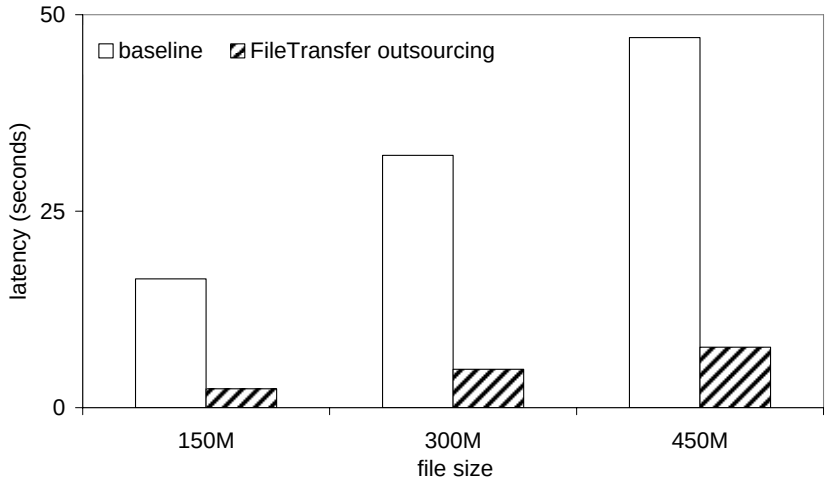


Figure 32 Latency Comparison for File Download

3.4.2. Inter-VM Communication Specialization

Kernel service outsourcing provides another opportunity that leads to performance improvement of inter-VM communication. If a socket connection is between two domains running in the same machine, the communication performance can be improved by using shared memory.

Previous works such as XWay [30], XenLoop [63], and XenSocket [67] have researched on accelerating inter-domain communication through shared memory. Our shared memory mechanism through kernel service outsourcing differs from them in several ways. First, our design requires no change in the API or ABI. Second, our mechanism is mostly implemented in user level, eliminating the need of modifying the underlying kernel for the implementation. Third, inter-domain communication is dynamically detected and switched to the special communication automatically. Also, the split model of frontend and backend agents allows heterogeneous guest OSes to

communicate through shared memory. Therefore, our mechanism can be regarded as a superset of other similar mechanisms.

Once a backend agent determines that a new TCP connection is destined to one of the VMs by monitoring the destination address of the `connect()` calls, the agent opens a shared memory channel with the backend agent of the destination domain and uses the channel for further communication. Figure 33 illustrates the specialized inter-VM communication through shared memory.

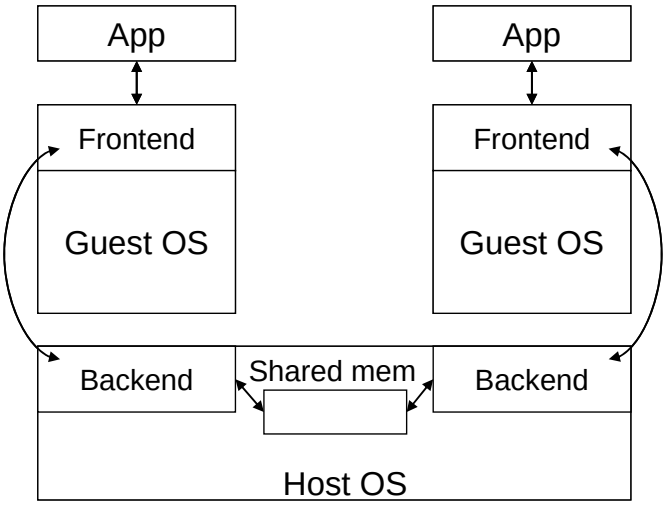


Figure 33 Specialization of Inter-VM Communications

To show the effectiveness of our optimization, we measured the network performance between two guest domains. In our experiments, we used a TCP/IP performance measurement tool [76], `iperf`, and measured the TCP throughput with various message sizes. Figure 34 presents maximum TCP throughput between a guest Windows and a guest Linux, and Figure 35 between two Windows guests. Network service outsourcing alone increases the inter-VM communication throughput, because network service outsourcing allows the packets to be processed only once in the host OS

kernel. Note that inter-VM packets are processed in each of two guest domains without network service outsourcing. Shared memory optimization improves the network performance further, increasing the maximum throughput by up to 8 folds compared to the baseline case, in which no kernel service outsourcing is used.

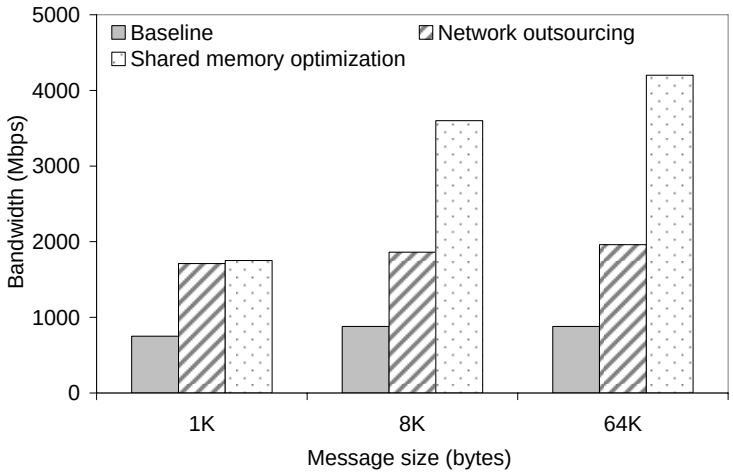


Figure 34 Inter-VM (Windows-Linux) TCP Throughput

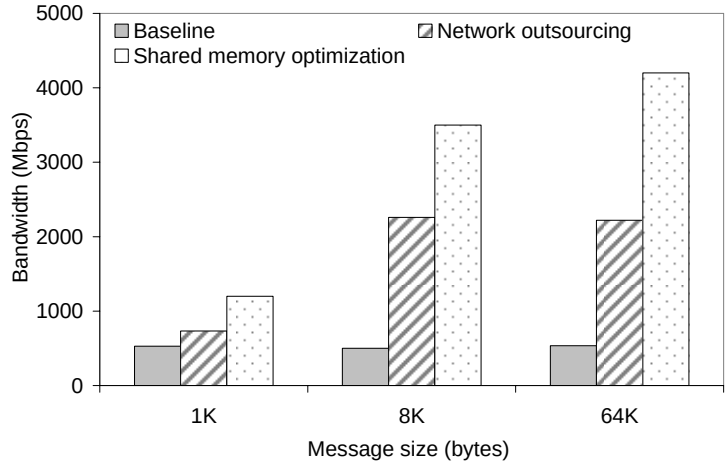


Figure 35 Inter-VM (Windows-Windows) TCP Throughput

Previous works such as XWay [30], XenLoop [63], and XenSocket [67] have researched on accelerating inter-domain communication through shared memory. Our shared memory mechanism through kernel service outsourcing differs from them in several ways. First, our design requires no change in the API or ABI. Second, our mechanism is mostly implemented in user level, eliminating the need of modifying the underlying kernel for the implementation. Third, inter-domain communication is dynamically detected and switched to the special communication automatically. Also, the split model of frontend and backend agents allows heterogeneous guest OSes to communicate through shared memory. Therefore, our mechanism can be regarded as a superset of other similar mechanisms.

CHAPTER 4

IMPROVING SYSTEM RELIABILITY THROUGH KERNEL SERVICE OUTSOURCING

4.1. Defense Based on Natural Diversity

Diversity is an important natural defense technique to increase the survivability of species in an epidemic outbreak, e.g., during the spread of a novel virus. Early attempts to apply diversity concepts such as N-version programming [4] for improving software reliability have encountered managerial problems such as high development costs and technical difficulties such as the overlap of fault modes among versions [33].

With the evolution of information technology, recent applications of diversity techniques have been more encouraging. For example, automated techniques (sometimes called artificial diversity [65]) have been successfully demonstrated to provide effective defenses against specific classes of software viruses. A concrete example is the use of ASLR (address space layout randomization) [68] in Linux and Windows Vista to defeat stack/buffer overflow attacks that rely on specific memory layout.

In contrast to artificial diversity, we explore the use of *natural diversity* among different currently existing operating systems (OS) such as Linux and Windows, to defeat attacks intended for one system but will not work on the others. Compared to artificial diversity, natural diversity has three major advantages. The first advantage of natural diversity is its effectiveness in defeating attacks that exploit vulnerabilities specific to an OS. The effective defense relies on the wide range of differences in kernel interfaces and

implementations of naturally diverse OSes such as Windows and Linux. These differences make it difficult for an attack to work simultaneously on two such naturally diverse OSes. The resulting combination achieves the software reliability advantages originally expected of N-version programming: independent failure modes among the versions.

The second advantage of natural diversity is low cost of both development and execution. On the software development side, the OSes are developed and maintained by different teams of programmers for different purposes. The Windows family of OSes has large commercial value. The Linux OS (and other open source software systems) represents the best efforts of the open source community. Consequently, natural diversity does not require additional programming costs. During execution, we show good performance in the outsourcing implementation of natural diversity, due to the better execution environment in the host OS for kernel services.

The third advantage of natural diversity is application-level backward compatibility. In our implementation, Windows applications execute without change in our environment using the Windows application programming interface (API), and reap benefits of increased software reliability and security from running Linux kernel services through the kernel service outsourcing method.

Despite these significant advantages, one potential concern with natural diversity is the relatively small number of variants being actively maintained. From our experience with two OSes, which is one of the simplest forms of natural diversity, we argue for its usefulness despite the minimal number of variants. We show that outsourcing is an effective defense against attacks that exploit vulnerabilities in specific guest kernel code components. Even though artificial diversity techniques can generate automatically many

variants, they are located in a restricted state space (e.g., ASLR on memory layout), which may be automatically and systematically explored by attackers who have sufficient resources. In comparison, the differences among the naturally diverse OSes extend from kernel interfaces to their implementations. Currently, there are very few known vulnerabilities shared by both Windows and Linux that can be exploited by a single attack. The long term software security offered by natural diversity is an interesting and open research challenge.

Our approach, kernel service outsourcing, achieves natural diversity by leveraging the recent virtual machine technology and software tools. The main idea of outsourcing [22] is to bypass some kernel services of a guest OS (e.g., Windows) and use equivalent, but different kernel services of the host OS (e.g., Linux). This way, a vulnerability in the guest OS (either because of the delay in applying patches or due to a completely new attack) would not succeed, since the vulnerable code is simply not executed. As a concrete example described in Section 4.2, a recent vulnerability, which could cause remote code execution, in Windows Vista TCP/IP protocol stack would be simply bypassed in our system that outsources guest the Windows network protocol stack with the Linux network protocol stack in the host OS.

In this study, we present a concrete feasibility demonstration of the natural diversity approach for improving software security and reliability. Our implementation combines a Windows guest OS with a Linux host OS, outsourcing TCP/IP network protocol stack and file systems. To the best of our knowledge, this implementation is the first feasibility demonstration of the natural diversity approach.

4.2. Effective Defense Through Kernel Service Outsourcing

Vulnerabilities found in the OS kernel can be particularly critical because they often lead to serious security holes such as remote code execution and denial of service. We have found that critical vulnerabilities still exist in modern operating system kernel from the Common Vulnerabilities and Exposures database [72].

Kernel service outsourcing bypasses vulnerabilities in certain parts of the kernel, preventing the attacks from exploiting the vulnerabilities in the specific kernel. For example, vulnerabilities in the Windows network stack can be defeated by outsourcing network service to the Linux kernel, because very few known vulnerabilities are shared by both Windows and Linux. We consider three cases of vulnerabilities in the TCP/IP stack for showing our effective defense through natural diversity.

4.2.1. TCP/IP Orphaned Connections Vulnerability

The TCP/IP implementation in certain versions of Windows operating systems, such as Windows XP, Vista Gold, SP1, and SP2, were not cleaning up state information properly. (CVE-2009-1926) To exploit the vulnerability, attackers could send specially crafted TCP/IP packets to the system that has a TCP/IP listening service, and cause TCP connections to stay indefinitely. As a result, the attackers could cause denial of service in the system.

To exploit this vulnerability, attackers need to establish a large number of connections closed by the application in the target system. Because of this reason, a potentially good target application for attackers is a web server. As an example, we show how attackers can exploit the vulnerability through a web service running in the target system.

1) The attacker establishes a TCP connection to the web server in the target machine with the advertized receive window size set to zero.

2) The attacker sends a GET request.

3) The web server application sends response data and closes the socket. Because the application closes the socket, the connection is now handled by the kernel.

4) The state of the connection moves to FIN_WAIT_1.

5) Because of the zero-sized window advertized by the attacker, the kernel sends the remaining data to the attacker one byte at a time, using TCP zero window probes.

6) When all the data are sent, the connection stays and is not cleaned up.

In step 6), the connection is supposed to be cleaned up and all previously allocated resources are required to be freed, when no more data is left to be sent. However, the mismanagement of the connection in the kernel code causes the connection to stay indefinitely. In consequence, the attackers can cause the system to consume unnecessary resources, eventually leading to denial of service in the system.

To identify the impact of the remaining TCP connections in the system, we measured the system memory used in one Windows XP system while we replay the attack from client machines. In Figure 36, we show our experimental results. The memory usage in the system under attack increases linearly as the number of hanging connections increases. In our experiments, we observed that around 35Mbytes of memory were consumed for every 10,000 remaining TCP connections.

Because the Linux kernel does not have the same kind of vulnerability, we can defend this attack by the network service outsourcing. First, we assign the target

machine's IP address to the host OS's network interface. Second, we start outsourcing the target machine's network service to the host OS by running frontend and backend agents. Finally, we restart the web server in the target machine.

Once network service outsourcing starts, the TCP connections are managed by the host OS. When we replayed the same attacks with zero-window size, we noticed no significant memory usage increase either in the target machine and the host OS, successfully nullifying the attacks from attacker machines.

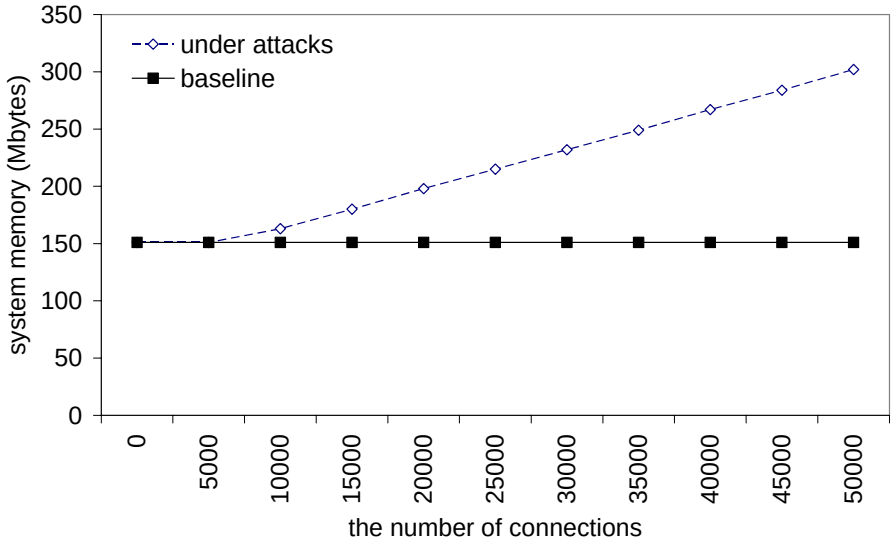


Figure 36 System Memory Consumption under DoS Attack

4.2.2. TCP/IP Timestamps Code Execution Vulnerbility

Another vulnerability found in the certain versions of Windows Vista is the TCP/IP Timestamps Code Execution Vulnerability (CVE-2009-1925). As described in Microsoft Security Bulletin (MS09-048), an error in the network stack implementation of the Windows kernel could cause misinterpretation of some data field as a function pointer.

Remote attackers could exploit this bug and execute arbitrary codes in the target machine by sending specially crafted packets to a TCP/IP listening service in the machine.

We were unable to replay this attack due to limited information. However, because this critical vulnerability is only to specific versions of Windows Vista, we believe that network service outsourcing would defeat the attacks exploiting this vulnerability.

4.2.3. NULL-pointer Dereference Vulnerability in udp_sendmsg()

One vulnerability found in the Linux network stack was from the UDP implementation in the Linux kernel 2.6.18 or earlier. The Linux kernel did not properly handle certain parameters of the `udp_sendmsg()` function. In consequence, a malicious attacker could send specially crafted commands to the system via a UDP socket involving the `MSG_MORE` flag, then, the user could gain privileges or cause denial of service by crashing the system due to NULL-pointer dereference. (CVE-2009-2698)

To exploit this vulnerability, the attacker needs to have a local account in the machine. Then, the vulnerability can be exploited as follows.

1) The attacker creates a UDP socket.
2) The attacker invokes the `sendto()` functions with `MSG_MORE` flag and other flags set properly.
3) The `sendto()` function eventually invokes the `udp_sendmsg()` function in the kernel.
4) In the `udp_sendmsg()` function, the `rt` routing table is initialized as NULL, but some code paths related the `MSG_MORE` flag leads to call the `ip_append_data()` function with the NULL `rt` pointer.

5) The `ip_append_data()` raises a NULL-pointer dereference error.

Through this NULL pointer dereference vulnerability, the attacker may have a chance to elevate its privilege or crash the system [72].

Because this vulnerability does not exist in the more recent Linux kernel (2.6.19 or later), network service outsourcing to the host OS kernel with version 2.6.19 or later can defend this vulnerability. In our experiments, we used the Linux kernel 2.6.9 for our guest machine, and our attack codes were able to cause the NULL-pointer dereference. However, the same attack was not successful when we applied network service outsourcing to the guest kernel and delegated the network service to the host OS, the kernel version of which was 2.6.24.1.

CHAPTER 5

A METHODICAL APPROACH FOR IMPLEMENTING KERNEL SERVICE OUTSOURCING

In previous chapters, we have shown that kernel service outsourcing is a powerful mechanism to improve the performance and the reliability of virtualized systems. However, because of many different types of kernel services existing in the operating systems and the great variety of operating systems and virtualization environments, applying the kernel service outsourcing mechanism to those environments becomes a difficult task. To meet these challenges, we present a methodical approach to implement kernel service outsourcing in heterogeneous, multi-kernel environments.

5.1. Four Steps to Implementing Kernel Service Outsourcing

The great variety of operating systems and virtualization environments prevents kernel service outsourcing from being easily applied. Different operating systems often have various types of system call interfaces, and the performance implication of each system varies because of different implementations of operating system kernels and virtual machine monitors (VMMs). To meet these challenges, we present a methodical, step-by-step approach by describing general steps of applying kernel service outsourcing in virtualized systems.

1) Identify overheads of a kernel service in virtualized systems
2) Intercept system calls related to the kernel service
3) Translate system calls to external kernel service calls

4) Develop run-time program modules for efficient inter-domain communication

In next subsections, we explain these steps in greater details.

5.1.1. Identify Overheads of a Kernel Service in Virtualized Systems

As a first step, we identify the overhead of kernel services in the guest systems. Kernel services with significant virtualization overhead are good candidates for kernel service outsourcing because the virtualization overhead often leads to poor performance and excessive resource usage for the kernel service processing.

As an example, in Figure 37, we compare the CPU usage of native systems and virtualized systems with two different workloads, network and disk writing. For the disk writing task, native Linux consumed around 40% of CPU, however, the guest Linux with para-virtualized devices (Linux+PVdev) needed around 62% of CPU for the same task. Similarly, with network workload, we observed that native Linux used only 12% of CPU to send network packets over 1Gbps Ethernet, while Linux+PVdev consumed around 90% of CPU.

Moreover, we found that the performance of disk and network decreased in virtualized systems. In Linux+PVdev, the disk write performance was 17.7Mbytes/sec, compared with 25.6Mbytes/sec in native Linux as shown in Figure 38. For more detailed performance comparison with network workload, refer to the previous chapters on network service outsourcing.

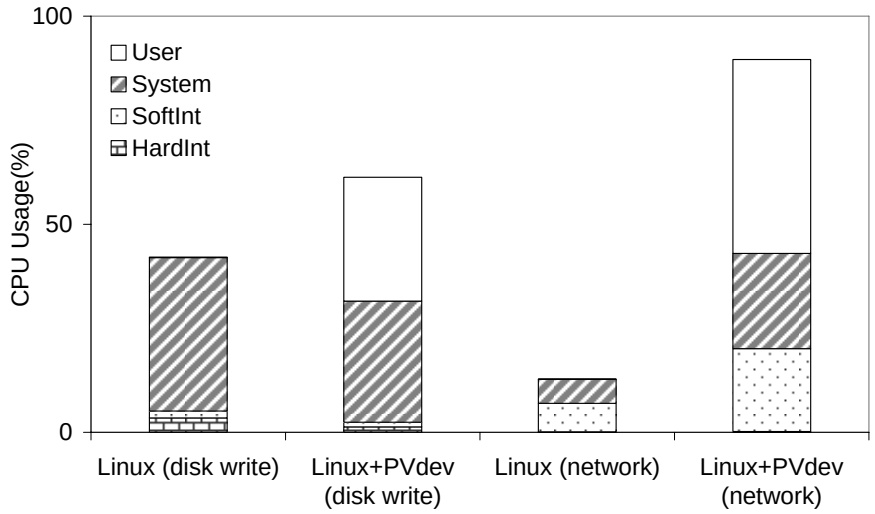


Figure 37 CPU Usage comparison of native systems and virtualized systems

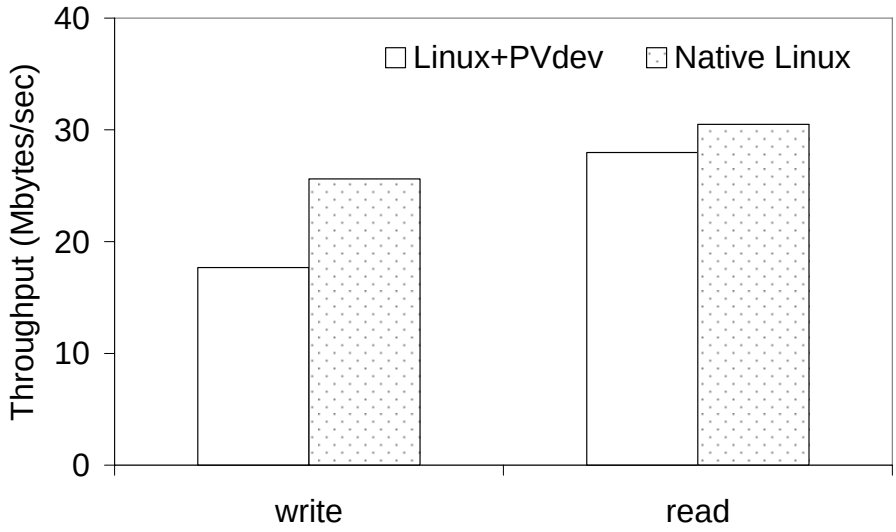


Figure 38 Disk performance comparison of native systems and virtualized systems

5.1.2. Intercept System Calls Related to the Kernel Service

Once we identify a kernel service that suffers from virtualization overhead, we determine which system calls that related to the kernel service are to be outsourced and be processed by an external kernel.

The network service from the OS kernel is a good candidate for kernel service outsourcing, because networking is relatively independent from other kernel services and

provides a clear interface (socket interface) for the network service. In order to outsource the network service, we intercept all the socket-related functions invoked by applications and delegate the processing of these functions to an external kernel. In our implementation of network service outsourcing, we intercepted 13 socket functions and 30 Winsock functions in guest Linux and Windows, respectively.

Another good candidate for kernel service outsourcing is the filesystem service. Intercepting filesystem-related system calls becomes more sophisticated than intercepting the network service system calls, because the filesystem-related system calls are shared by other kernel services. For example, the `open()` function can be used not only for opening a file but for opening other resources such as Unix pipe and shared memory. In order to outsource only file-related system calls, we inspect the function arguments and track file handles in order to determine if the function is file-related (i.e. to be outsourced) or not (i.e. to be processed by the guest kernel).

5.1.3. Translate System Calls into External Kernel Service Calls

To bypass and delegate the processing of a kernel service, we intercept the system calls and redirect them to an external kernel. However, local kernels and external kernels often have different implementations, thus, we need to translate the system calls to make two different interfaces compatible.

The translation process involves the conversion of data structures, constant values, function behaviors, and different extension functions.

Data structures. Some data structures have different definitions in different implementations. For example, `struct fd_set` used in the `select()` function have different structure definitions in Windows and Linux.

Constant values. Constant values such as error codes may vary; an error code for refused connection in Linux is 111, while the error code in Windows is 10061.

Function behaviors. The behaviors of some functions are different. In our experiments, for example, we observed that `send()` in the Linux implementation sometimes returns with partially sent results when the sending buffer is almost full. However, the Winsock implementation either sends the requested message entirely or returns an error.

Different extensions. Each implementation has its own extensions other than standard socket API functions. For example, Linux provides `sendfile()` for efficient data transfer between a file and a socket. Winsock provides functions such as `ConnectEx()`.

We translate the system calls into an intermediate interface. As an example, in Table 12, we present an intermediate interface for filesystem service outsourcing. Currently, our implementation only supports basic file operations such as read and write. We plan to support for full filesystem outsourcing in the future.

Table 12 Interface for File-system Service Outsourcing

Name	*Description*
`EX_openfile`	Creates a file descriptor
`EX_closefile`	Closes a file descriptor
`EX_readdir`	Reads a directory
`EX_getattr`	Returns information related a file
`EX_readfile`	Reads the content of a file
`EX_writefile`	Writes the content of a file
`EX_truncate`	Truncates a file

5.1.4. Develop Run-time Program Modules for Efficient Inter-domain Communication

The final step of implementing kernel service outsourcing is the development of run-time program modules that establish efficient inter-domain communication between a local kernel and an external kernel.

We develop two run-time program modules, frontend agent and backend agent in a local kernel and an external kernel, respectively. Both modules can be either a user-level program or a kernel-level one. We currently implement them in user-level for easier maintenance in our implementations.

The frontend agent intercepts the system calls to the I/O subsystem we outsource, translates the function calls to the intermediate interface, redirects them to the backend agent, and monitors asynchronous events from the backend agent.

The backend agent, running in the external kernel, processes the requests from the frontend agent, keeps the states of the outsourced kernel service, and delivers asynchronous events to the frontend agent for notification.

For efficient communication between the frontend and the backend agent, VMM support is often required. We described how we used the common facilities of VMM to implement the communication channel between two agents in Chapter 3.

5.2. Evaluating the Performance Impact of Kernel Service Outsourcing

We have described our methodical, step-by-step approach to implement kernel service outsourcing for such kernel services as network and filesystem in different combinations of guest OSes. For evaluation, we measure network packet processing latency in each layer of the guest and host kernels in order to see how kernel service

outsourcing reduces the kernel processing overhead during network packet transmission. Moreover, we monitor low-level system characteristics using various tools to collect system resource usage information and hardware counters during I/O operations in virtualized systems, and explain the performance impact of kernel service outsourcing.

5.2.1. A Layered Analysis of Packet Processing

For understanding the overhead of packet processing in different layers of virtualized systems, we measured the time a UDP packet spent in each layer. In our experiments, we sent a UDP packet of 2Kbytes from a user-level application and measured latency per layer by kernel instrumentation and time-stamping the packet at each layer. We illustrate the per-layer latency measured in native Linux, guest Linux with para-virtualized device (Linux+PVdev), and guest Linux with network outsourcing (Linux+Outsourcing) in Figure 39.

The overall latency for the UDP packet reaching a physical NIC from the application was 6.5us and 18.3us for native Linux and Linux+Outsourcing, respectively. Surprisingly, the UDP packet latency for Linux+PVdev was around 4ms, which was extremely larger than those of other cases. This huge latency comes from the design of KVM para-virtualized network device (Virtio), which uses a timer to batch network packet transmission. Because the timer resolution is 4ms, the UDP packet is delayed by that amount of time.

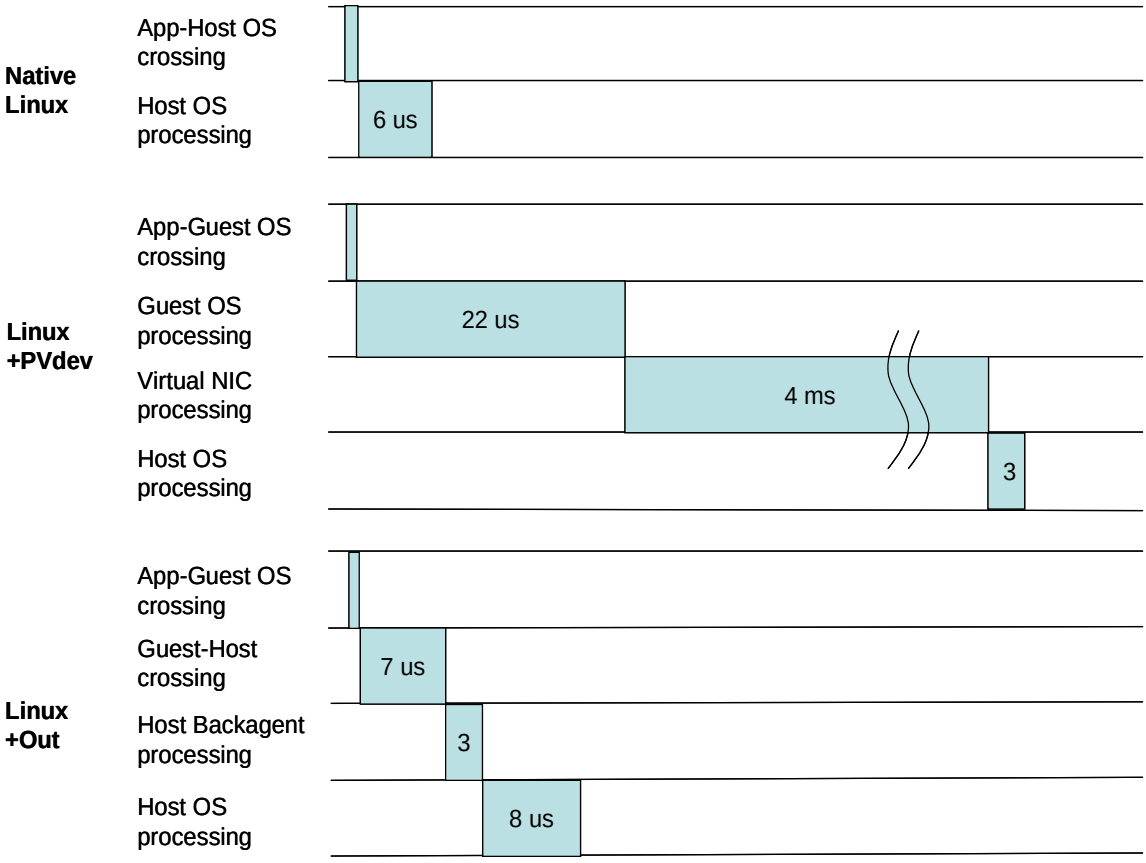


Figure 39 Per-layer latency for native Linux, Linux+PVdev, and Linux+Out

In native Linux, network processing for a UDP packet in the network stack was around 6us, but in Linux+PVdev, the network processing time in the guest kernel increased to 22us. On the other hand, in Linux+Out, because the network processing in the host kernel is much faster (8 us) than that in the guest (22 us), we can see that the overall latency for sending a UDP packet is smaller in Linux+Out than that in Linux+PVdev, despite the overhead of guest-host crossing and the backagent processing in the host kernel.

5.2.2. Monitoring Low-level System Characteristics

5.2.2.1. Tools for Low-level System Characteristic Measurement

We used three tools to measure low-level system characteristics and to see how they change with kernel service outsourcing during I/O operations such as network packet transmission in guest OSes.

Dstat. Dstat is a resource statistics tool that collects system resource information such as CPU usage, the number of interrupts generated in the system, the number of context switches, and the amount of data processed in disk and network devices. During our experiments, we monitor resource statistics using dstat with the interval of one second.

Kvm_stat. Kvm_stat is a performance monitoring tool that collects statistics specifically related to virtual machines and the KVM module in the host OS kernel. The system performance numbers collected by kvm_stat include the followings.

Table 13 System counters collected by kvm_stat

Name	Description
VMexit	the counter for the `VMEXIT` instructions
Insn_emulation	the number of instructions emulated by the host OS on behalf of the guest OS
Io_exits	the number of occasions the guest OS exits because of an IRQ
Irq_injections	the number of IRQs raised to the guest OS

OProfile. OProfile is a system-wide profiling tool for Linux systems. OProfile collects various system statistics by leveraging hardware counters. OProfile can not only profile user-level programs but also kernel-level software routines such as kernel routines,

kernel modules, and hardware and software interrupt handlers. Table 14 lists some of the hardware counter events OProfile can collect.

Table 14 Hardware counter events monitored by OProfile

Event type	Description
LLC_MISSES	L2 cache misses
DTLB_MISSES	DTLB misses
BUS_TRAN_MEM	The number of memory bus transactions
INST_RETIRED	The number of instructions retired

5.2.2.2. Experimental Setup

Our experimental machine has an Intel Core 2 Duo 2.4Ghz processor with a 4MB L2 cache, 2 GB of main memory and a gigabit network adapter card. The processor has two CPU cores, but we turned one CPU core off to eliminate the scheduling effect and monitor system resource usage more accurately. We used the Linux kernel version 2.6.25 for the host kernel and KVM version 84.

We used two different guest OSes, Linux and Windows for our experiments. For the Linux guest, we installed the Debian/Linux distribution with the kernel version 2.6.25. For the Windows guest, Windows XP with service pack 2 was used in our experiments.

In order to generate network workload, we used the iperf TCP/UDP bandwidth measurement tool. For disk workload, we wrote a simple benchmark program that reads from and writes to disk in order to generate disk I/O operations. Note that we used O_DIRECT flag when our disk application opens a test file to eliminate the buffer cache effect in the operating systems.

Low-level system performance counters are collected during the time the network and disk benchmark programs are running. We used three low-level system performance counter tools, dstat, kvm_stat, and OProfile, described in the previous subsection.

In next subsections, we present our measurement results for low-level system characteristics and describe how these results explain the performance gain the kernel service outsourcing mechanism brings.

5.2.2.3. CPU Usage

Dstat collects the CPU usage for five different categories, User, System, Wait, SoftInt, and HardInt. “User” denotes the CPU usage from user-level processes. Thus, the CPU usage from the guest OS kernel is included in the User category. “System” denotes the CPU used by the host kernel, “SoftInt” the CPU used for software interrupt handling. “HardInt” measures the CPU usage for hardware interrupt processing. In our workload, however, only slight amount of CPU resource was used for hardware interrupt processing, therefore, we do not include the category in our graph. “Wait” denotes the CPU time of waiting for I/O operations to be completed. Because CPU is simply waiting during that time, we do not include this category of CPU usage for our analysis.

Figure 40 presents the CPU usage with the network workload generated by iperf. We monitored the overall CPU usage in two different guest OSes, Linux and Windows, with two network I/O mechanisms, para-virtualized device (Linux+PVdev and Win+PVdev) and network service outsourcing (Linux+out and Win+out). Note that iperf was able to reach the maximum throughput of 940Mbps with Linux+PVdev, Linux+out, and Win+out, but Win+PVdev achieved only 304Mbps for its maximum network throughput.

Comparing the CPU usage of Linux+PVdev and Linux+out, we observed that network service outsourcing significantly reduced the overall CPU usage from 88% to 32% for the same network performance. Particularly, the CPU usage for the User category was decreased from 47% to 10% with network service outsourcing, because most of guest network processing was bypassed and delegated to the host kernel with kernel service outsourcing.

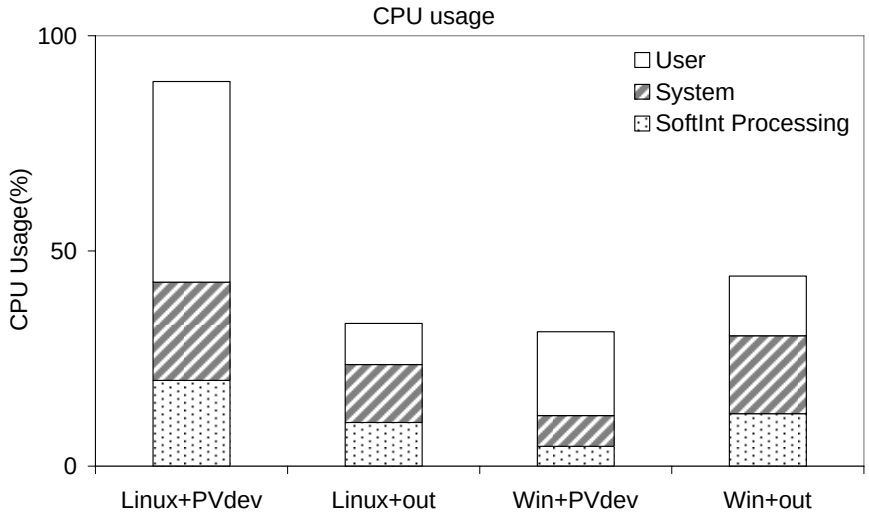


Figure 40 Overall CPU Usage with network workload

In the Windows guest, 31% of CPU resources were used for reaching 304Mbps with para-virtualized devices. On the other hand, network outsourcing used 43% of CPU to achieve the network throughput of 940Mbps. Therefore, our measurement reveals that network outsourcing was able to transmit the same amount of data with less than half of the CPU resource required with para-virtualized devices.

5.2.2.4. Emulated Instructions

Because our guest OSes are fully virtualized, some of the native instructions executed by the guest kernel must be emulated by the VMM. This emulation causes some overhead in the guest kernel processing, therefore, we also measure and compare the number of emulated instructions with para-virtualized devices and outsourcing.

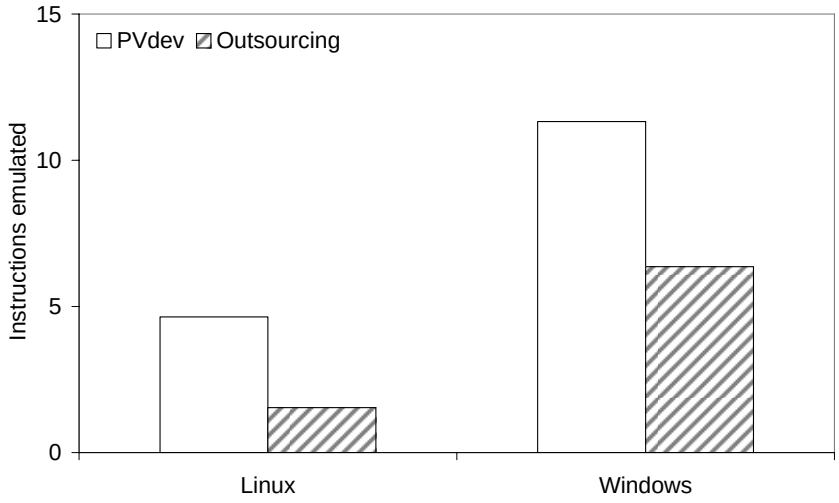


Figure 41 The number of instructions emulated during 1Mbit network transmission

Figure 41 presents the number of emulated instructions for transmitting the data of 1Mbits over network. From the measurement results, we observed that network outsourcing significantly reduced the number of emulated instructions compared with para-virtualized devices. In the guest Windows, we found that more instructions needed to be emulated than in the guest Linux. We believe that this higher emulation cost causes the greater CPU usage in Win+Out than in the Linux+Out as shown in Figure 40.

5.2.2.5. Context Switches

Another system characteristic we found interesting in our experiments is the number of context switches. We show the measurement results of the number of context

switches for transmitting 1Mbits over network in Figure 42. Note that Linux+PVdev causes significantly more context switches than network outsourcing. For example, in the guest Windows, context switches of PVdev during 1Mbps data transmission are nearly 10 times more frequent than those of network outsourcing.

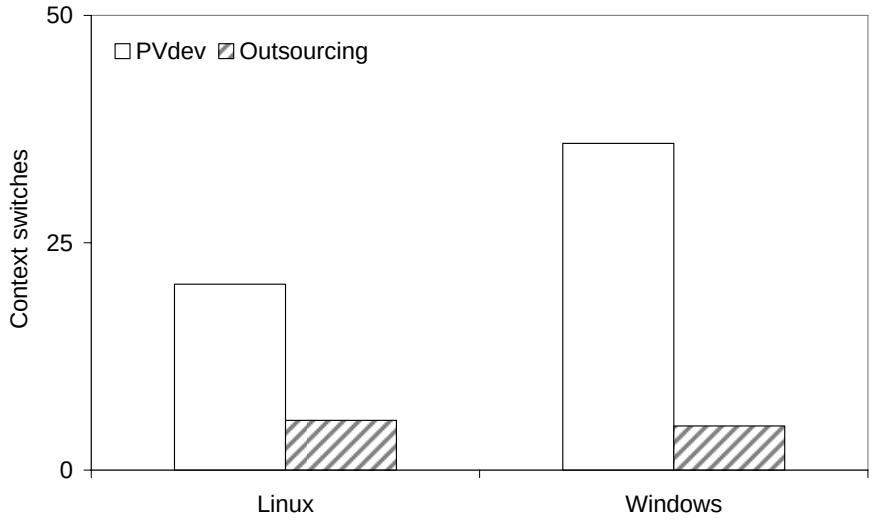


Figure 42 The number of context switches during 1Mbit network transmission

We believe that the frequent context switches with PVdev are from the overheads of packet segmentation and frequent data transfers between frontend and backend drivers. Some researchers found the similar overhead in the Xen hypervisor [43]. On the other hand, network outsourcing eliminates the packet segmentation by transferring packet payload to the host OS with a high-level interface, socket API. Transferring large packets without segmentation reduces the number of frequent data exchange between the guest OS kernel and the host kernel, subsequently decreasing the number of context switches. This performance benefit of using high-level interfaces and large packet transfer is similar to that of the optimization with TCP segmentation offload (TSO) in [43].

5.2.2.6. OProfile Results

We also measured low-level hardware counters such as the number of instruction executed and cache misses using OProfile. In this particular study, we collected the number of instructions executed, memory bus transactions, L2 misses, and TLB misses.

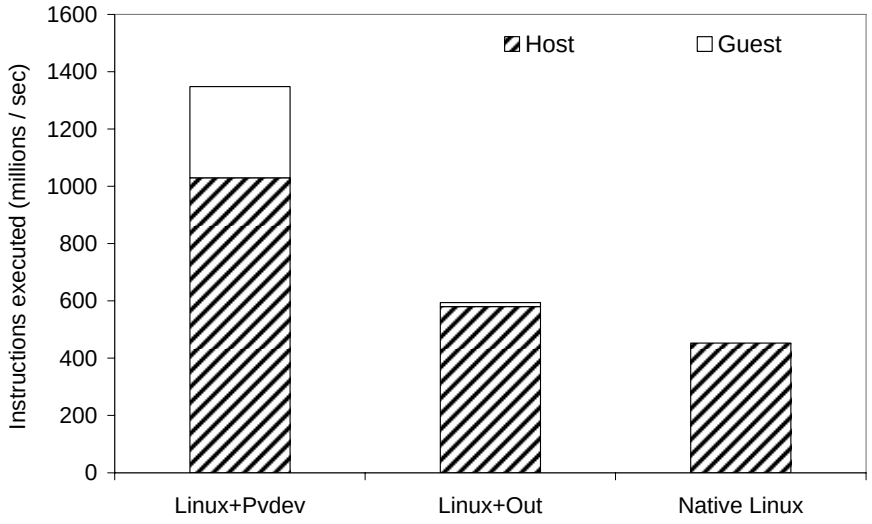


Figure 43 The number of instructions executed with network workload

In Figure 43, we present the number instructions executed in the guest and host, respectively. With our measurement results, we observed that Linux+Out reduced the number of executed instructions to less than half compared with Linux+PVdev. This reduced number of instructions of network outsourcing saves CPU resources, leading to better network performance in higher bandwidth environments such as 10Gbps Ethernet. Note that Linux+Out eliminates most of processing in the guest, because network outsourcing bypasses guest network stack codes.

We present memory-related hardware counters measured with network workload in Figure 44. Note that we normalized the counter values to the Linux-PVdev case so that

we can compare the changes in the values with network outsourcing and with native OS. While L2 cache misses and TLB misses were smaller with network outsourcing, the number of memory bus transactions was nearly identical in all three cases. We believe that memory bus transactions occur when network packets are transferred to the physical NIC, therefore, this number corresponds to the maximum network throughput. Note that the maximum network throughputs in Linux+PVdev, Linux+Out, and native Linux were all reaching to 940Mbps.

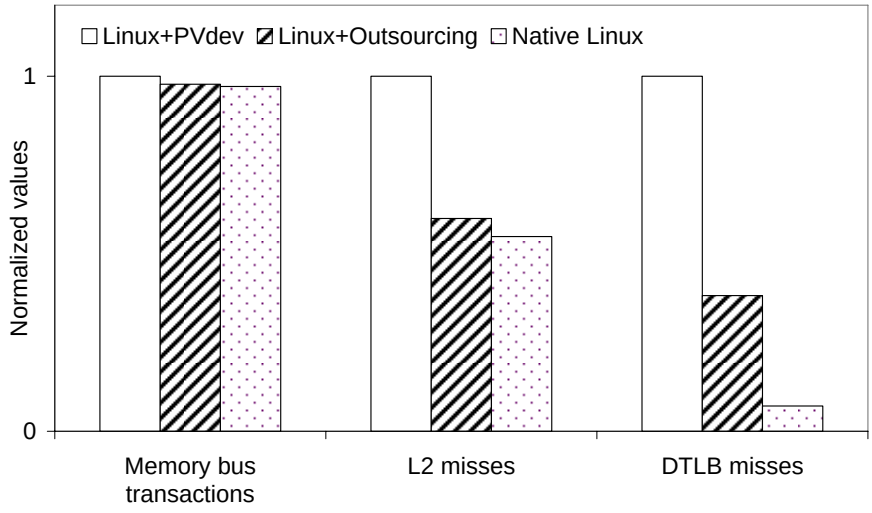


Figure 44 Memory-related hardware counters with network load

CHAPTER 6

CONCLUSION

6.1. Summary

Virtualization techniques are being widely used in data centers for high resource utilization and easy maintenance of computing resources through running multiple guest OSes concurrently on a single physical host. In virtualized environments, different types of guest OS kernels with different functionalities may exist simultaneously. This dissertation explored the opportunity to improve system performance and reliability through kernel service outsourcing that delegates the services of one guest kernel to another OS kernel in the system.

First, we have shown that kernel service outsourcing can achieve significantly better performance than native guest OS execution in several important cases. Using different guest OSes such as Linux and Windows, we observed that the throughput performance of network and filesystems with kernel service outsourcing increased several times compared with the performance of guest OS processing with para-virtualized devices. These performance gains come from more efficient kernel processing of privileged OS kernels such as the host OS kernel than that of guest OS kernels.

In addition to performance measurement, we monitored low-level system characteristics and observed the impact of bypassing guest processing. We found that delegation of kernel processing to the privileged kernels reduced CPU usage significantly by bypassing slow processing in the guest OS kernel. Our measurement results with low-

level system characteristics, such as the number of context switches and the number of emulated privileged instructions, explain how kernel service outsourcing eliminates overhead and achieves improved performance in virtualized systems.

Second, we have described how kernel service outsourcing can be used to create an instance of natural diversity, which improves system software security and reliability. Our implementation combining a Windows guest OS with a Linux host OS, outsourcing the TCP/IP network protocol stack successfully defended malicious attacks that targeted the implementation-dependant vulnerabilities in the network protocol stack. To the best of our knowledge, this implementation is the first demonstration of feasibility of the natural diversity approach.

Natural diversity has several advantages. First, it is an effective defense against attacks targeting specific OS component vulnerabilities. We demonstrate this by our success in defending concrete real attacks targeting network protocol stacks. Second, this natural diversity has low development costs (Windows and Linux are maintained by teams other than the authors) and low execution penalties. Third, natural diversity supports backward compatibility at the application level. In our implementation, Windows applications ran unmodified in the Windows guest OS and received the benefits of natural diversity through kernel service outsourcing that executed kernel services in the Linux host OS.

6.2. Future Work

Our approach of kernel service outsourcing has some limitations. First, the mechanism to invoke a system call to an external kernel is more sophisticated than the system call of a local guest kernel system. Therefore, kernel services that require frequent

system call invocations benefit less from I/O service delegation. For example, if a kernel service uses the system cache frequently, outsourcing the kernel service would actually affect performance adversely because of the cost of the sophisticated mechanism of system call invocation. Which kernel service in the OS kernel to choose as a candidate for kernel service outsourcing is an interesting question for future research. Second, an external OS kernel that provides kernel services to guest OSes needs to keep states for each guest OS when multiple guest OSes outsource their kernel processing to the external kernel. In the event of failure of the external kernel, the states kept in the external service could become a single point for terminating kernel services for guest OSes that use kernel service outsourcing. We leave to future work these challenges about the replication and restoration of states after such a failure in the external kernel. Third, any application that does not use the standard API for I/O operations cannot get the benefit of kernel service outsourcing. Application of the kernel service outsourcing mechanism to non-standard APIs is a subject for future research.

When multiple guest OSes use kernel service outsourcing, scheduling kernel services among guest OSes could affect the performance of each guest OS. We leave to our future work the research on these challenges concerning the fairness and efficient scheduling of guest services in the external kernel.

We believe that kernel service outsourcing can be used for other purposes than improved performance and reliability in virtualized systems. For example, kernel service outsourcing can be used to speed up the encryption/decryption process by using a special CPU core dedicated to another domain. Using kernel service outsourcing with special hardware is our future research subject.

Our study has shown that natural diversity is a viable approach for improving system software security and reliability. Although our approach is limited to the combination of two different OSes, one as guest OS and the other as host OS in a virtualized environment, our results suggest that further research on natural diversity is warranted. Future work on natural diversity includes challenges in implementation techniques (e.g., execution overhead due to a level of indirection) and limitations of applicability (e.g., characterization of security and reliability problems that can be solved by natural diversity). These interesting challenges show that natural diversity represents a promising new approach to improved system software security and reliability.

REFERENCES

[1] M. Abbott and L. Peterson, “Increasing Network Throughput by Integrating Protocol Layers”, *IEEE/ACM Transactions on Networking*, vol. 1, pp. 600-10, 1993.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. “Mach: a New Kernel Foundation for UNIX Development”, In *Proceedings of the USENIX Conference*, pages 93–112, July 1986.

[3] D. Aloni: "Cooperative Linux", In *Proceedings of the Ottawa Linux Symposium (OLS-2004)*, pp.23-31 (2004).

[4] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Execution," In *Proceedings of the 1st IEEE International Computer Science Applications Conference*, IEEE Press, 1977, pp. 149–155

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, and R. Neugebauer. “Xen and the art of virtualization”. In *Proceedings of the ACM Symposium on Operating Systems Principles*. Oct. 2003.

[6] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, “The multikernel: a new OS architecture for scalable multicore systems” In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, Montana, 2009.

[7] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. “Operating System Support for Planetary-Scale Network Services”. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implmentation (NSDI)*. San Francisco, CA, 2004.

[8] A. Bavier, T. Voigt, M.Wawrzoniak, and L. Peterson. “SILK: Scout Paths in the Linux Kernel”, Technical Report 2002–009, Uppsala University, Sweden, February 2002.

[9] F. Bellard: "QEMU, a Fast and Portable Dynamic Translator", In *Proceedings of the USENIX 2005 Annual Technical Conference*, FREENIX Track, pp. 41-46 (2005).

[10] S. Bhatia, C. Consel, A. Le Meur, and C. Pu. “Automatic Specialization of Protocol Stacks in OS Kernels”. In *Proceedings of the 29th IEEE Conference on Local Computer Networks*, Tampa, Florida, November 2004.

[11] S. Bhatia, C. Consel, and C. Pu. “Remote Customization of Systems Code for Embedded Devices”, In *Proceedings of the Fourth ACM International Conference on Embedded Software*, Pisa, Italy, September 2004.

[12] G. Bryce, G. Muller, "Matching Micro-kernels to Modern Applications Using Fine-grained Memory Protection", In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, p.272, October 25-28, 1995.

[13] K. Buchacker and V. Sieh. “Framework for Testing the Fault-tolerance of Systems Including OS and Network Aspects”. In *Proceedings of the IEEE Symposium on High Assurance System Engineering*, pages 95–105, October 2001.

[14] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. “Disco: Running commodity operating systems on scalable multiprocessors”. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 1997.

[15] E. Cecchet, J. Marguerite, and W. Zwaenepoel: "Performance and scalability of EJB applications", In *Proceedings of the ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 246-261 (2002).

[16] J. Chu. “Zero-copy TCP in Solaris”. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, 1996.

[17] D. Clark and D. Tennenhouse. “Architectural Considerations for a New Generation of Protocols”. In *Proceedings of the SIGCOMM*, Philadelphia, Pennsylvania, Sept. 1990.

[18] J. Dike. “A User-mode Port of the Linux Kernel”. In *Proceedings of 5th Annual Linux Showcase & Conference*, Oakland, CA, 2001.

[19] A. Dinaburg, P. Royal, M. Sharif, and W. Lee: "Ether: malware analysis via hardware virtualization extensions", In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp.51-62, 2008.

[20] P. Druschel and L. Peterson, “Fbufs: A high-bandwidth cross-domain transfer facility”. In *Prcdeedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 1993.

[21] H. Eiraku and Y. Shinjo, **“**Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions”. In *Proceedings of the USENIX BSDCon*, pp. 91-102, 2003.

[22] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato. “Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments”. In *Proceedings of 2009 ACM Symposium on Applied Computing (SAC 2009),* March 2009.

[23] B. Ford, J. Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model", In *Proceedings of the Usenix Winter Conference*, California, United States, January 1994, pages 97-114

[24] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey: "Server Network Scalability and TCP Offload", In *Proceedings of the USENIX Annual Technical Conference*, pp. 209-222, 2005.

[25] T. Garfinkel and M. Rosenblum: "A Virtual Machine Introspection Based Architecture for Intrusion Detection", In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, 2003.

[26] A. Gavrilovska, S. Kumar, S. Sundaragopalan, and K. Schwan, "Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications", In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2005

[27] R. Goldberg. "Survey of Virtual Machine Research", *IEEE Computer*, pages 34-45, June 1974.

[28] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, and N. Tolia, "GViM: GPU-accelerated Virtual Machines", In *Proceedings of the 3rd Workshop on System-level Virtualization for High Performance Computing*, 2009.

[29] K. Kant: "TCP Offload Performance for Front-End Servers", In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM 03)*, pp.3242-324 (2003).

[30] K. Kim, C. Kim, S. Jung, H. Shin, and J. Kim: "Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen", In *Proceedings of the International Conference on Virtual Execution Environments (VEE-08)*, 2008.

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

[32] A. Kivity, Y. Kamay, and D. Laor: "KVM: the Linux Virtual Machine Monitor", In *Proceedings of the Linux Symposium*, pp.225-230, 2007.

[33] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming.", IEEE Transactions on Software Engineering 12, pp 96-109, 2008.

[34] Y. Koh, C. Pu, S. Bhatia, and C. Consel: "Efficient Packet Processing in User-Level OSes: A Study of UML", In *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN)*, 2006.

[35] Y. Koh, C. Pu, Y. Shinjo, H. Eiraku, G. Saito, D. Nobori, "Improving Virtualized Windows Network Performance by Delegating Network Processing", In *Proceedings of the IEEE Conference on Network Computing and Applications (NCA)*, 2009.

[36] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan, “C-Core: Using Communication Cores for High Performance Network Services”, In *Proceedings of the IEEE Conference on Network Computing and Applications (NCA)*, 2005.

[37] D. Laor, “KVM PV Devices”, Qumranet, 2007.

[38] J. Liedtke, " On Micro-Kernel Construction ", In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[39] L. Litty: "Hypervisor Support for Identifying Covertly Executing Binaries", In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[40] J. Liu, W. Huang, B. Abali and D.K. Panda. “High Performance VMM-Bypass I/O in Virtual Machines”. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, May 2006

[41] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, and P. Wagle, C. Consel, G. Muller, and R. Marlet. “Specialization Tools and Techniques for Systematic Optimization of System Software”, *ACM Transactions on Computer Systems*, Vol. 19, No. 2, pp 217-251, May 2001.

[42] L. McVoy and C. Staelin. “lmbench: Portable tools for performance analysis”. In *Proceedings of the USENIX Annual Technical Conference*, pages 279-294, Berkeley, 1996.

[43] A. Menon, A. Cox, and W. Zwaenepoel: "Optimizing Network Virtualization in Xen", In *Proceedings of the 2006 USENIX Annual Technical Conference*, pp.15-28, 2006.

[44] D. Mosberger and T. Jin. “httperf: A Tool for Measuring Web Server Performance”, Performance Evaluation Review, Volume 26, Number 3, December 1998, 31-37.

[45] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. “Analysis of Techniques to Improve Protocol Processing Latency”. In *Proceedings of the ACM SIGCOMM*, pages 73-84, Stanford, California, August 1996.

[46] E. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, “Helios: Heterogeneous Multiprocessing with Satellite Kernels”, In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP09)*, Big Sky, Montana, October 2009

[47] B. Pfaff, T. Garfinkel, and M. Rosenblum, “Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks”, In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[48] D. Price and A. Tucker. “Operating System Support for Consolidating Commercial Workloads”. In *Proceedings of the 18th Large Installation System Administration Conference (LISA)*, Atlanta, GA, 2004.

[49] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole and K. Zhang. “Optimistic Incremental Specialization: Streamlining a Commercial Operating System”, In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[50] C. Pu, H. Massalin, and J. Ioannidis. “The Synthesis Kernel”, Computing Systems 1, pp. 11-32, Winter 1988.

[51] H. Raj and K. Schwan, “High Performance and Scalable I/O Virtualization via Self-Virtualized Devices”, In *Proceedings of the ACM Internal Symposium on High Performance Distributed Computing (HPDC)*, 2007.

[52] H. Raj and K. Schwan, “O2S2: Enhanced Object-based Virtualized Storage”, ACM Operating Systems Review, October 2008.

[53] R. Riley, X. Jiang, and D. Xu: "Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing", In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[54] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, and W. 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.

[55] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong: "TCP Onloading for Data Center Servers", IEEE Computer, pp. 48-58 (2004).

[56] J. S. Robin and C. E. Irvine: "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", In *Proceedings of the USENIX Security Symposium*, 2000.

[57] M. Rosenblum and T. Garfinkel: "Virtual Machine Monitors: Current Technology and Future Trends", IEEE Computer, Vol. 38, No. 5 pp. 39-47, May 2005.

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

[59] A. Seshadri, M. Luk, N. Qu, , and A. Perrig: "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes", In *Proceedings of ACM Symposium on Operating Systems Principles*, pp.335-350, 2007.

[60] S. Son, J. Kim, E. Lim, and S. Jung: "SOP: A Socket Interface for TOEs", Internet and Multimedia Systems and Applications, pp. 294-299, 2004.

[61] J. Sugerman, G. Venkitachalam, and B. Lim: "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[62] C. Waldspurger. “Memory and Resource Management in VMWare ESX Server”, In *Proceedings of the USENIX Synposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[63] J. Wang, K. Wright, and K. Gopalan: "XenLoop: A Transparent High Performance Inter-VM Network Loopback", In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC)*, 2008.

[64] A. Whitaker, M. Shaw, and S. Gribble. “Scale and Performance in the Denali Isolation Kernel”. In *Proceedings of the USENIX Synposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[65] D. Williams, Wei Hu, J.W. Davidson, J.D. Hiser, J.C. Knight and A. Nguyen-Tuong, “Security through Diversity: Leveraging Virtual Machine Technology”, IEEE Security and Privacy, (7):1, pp 26-33, 2009.

[66] M. Xu, R. Sandhu, X. Jiang, and X. Zhang, “Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection”, In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pp.71-80, 2007.

[67] X. Zhang, S. McIntosh, P. Rohatgi, J. L. Griffin, "XenSocket: A High-Throughput Interdomain Transport for Virtual Machines", In *Proceedings of the Middleware Conference*, 2007.

[68] Address Space Layout Randomization (http://pax.grsecurity.net/docs/aslr.txt), September 2009.

[69] Apache Web Server (http://www.apache.org), March 2008.

[70] Bochs: The Cross Platform IA-32 Emulator (http://bochs.sourceforge.net), May 2006.

[71] Cygwin (http://www.cygwin.com), May 2006.

[72] Common Vulnerabilities and Exposuers (http://cve.mitre.org), September 2009.

[73] Dokan: user mode file system for Windows (http://dokan-dev.net/en), October 2009.

[74] FUSE: Filesystem in Userspace (http://fuse.sourceforge.net), October 2009.

[75] Httperf, a web server performance measurement tool (http://sourceforge.net/projects/httperf), May 2006.

[76] Iperf (http://sourceforge.net/projects/iperf), March 2008.

[77] Linux VServers Project. (http://linux-vserver.org/), May 2006.

[78] Microsoft Developer Network (http://msdn.microsoft.com), March 2009.

[79] QEMU, an open-source processor emulator (http://www.nongnu.org/qemu), March 2009.

[80] User-Mode Linux kernel (http://user-mode-linux.sourceforge.net/), May 2006.

[81] VMware: Virtual Infrastructure Software. (http://www.vmware.com), May 2006.

[82] The Wine Project (http://www.winehq.org), May 2006.

[83] Para-virtualized Windows driver for Xen (http://wiki.xensource.com/xenwiki/XenWindowsGplPv/), March 2009.