RICE UNIVERSITY

Optimizing Network I/O Virtualization through Guest-Driven Scheduler Bypass

by

Joanna Crompton

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Alan L. Cox
Alan L. Cox, Chair
Associate Professor of Computer Science
and Electrical and Computer Engineering

Scott Rixner
Associate Professor of Computer Science
and Electrical and Computer Engineering

David B. Johnson
Professor of Computer Science and
Electrical and Computer Engineering

HOUSTON, TEXAS

June, 2010
ABSTRACT

Optimizing Network I/O Virtualization through Guest-Driven Scheduler Bypass

by

Joanna Crompton

Virtualization is increasingly utilized for consolidating server resources to improve efficiency by conserving power and space. However, significant hurdles remain in achieving satisfactory performance in a virtualized system. Notably, virtualization of network I/O continues to be a performance barrier. The driver domain model of I/O virtualization suffers from an inherent network performance disadvantage due to the necessity of scheduling a driver domain. However, this virtualization model is desirable because of its fault tolerance and isolation properties. In this work, I argue that it is possible to overcome the barrier of network I/O performance while maintaining domain protection by providing a worldswitch mechanism which enables guests to operate the driver domain on their own behalf without the intervention of the scheduler. I describe my implementation of the worldswitch mechanism and evaluate its performance. I show that with the worldswitch enabled, guests achieve higher bandwidth and lower latency than in an unmodified system.
Acknowledgments

I would like to express my sincere thanks to my advisors, Dr. Alan Cox and Dr. Scott Rixner. Without their patient support, motivation, and guidance, this thesis would not have been possible.

I thank my undergraduate advisors, Dr. Steven Benzel and Dr. Nadeem Hamid, for their confidence and support. It was their example that inspired me to enter graduate school.

I thank all my Rice colleagues and friends for their constant encouragement, as well as for valuable criticism and suggestions. I am especially indebted to all those who kept me supplied with caffeine and those who offered to dance through my defense.

Finally, I would like to thank Kristina Andriana Koutsoudas and all her students for providing much-needed relaxation and optimism. Their works create an unremitting joy that they were willing to share with me throughout the creation of this thesis.
Contents

Abstract

Acknowledgments

List of Illustrations

1 Introduction

1.1 Contributions .............................................. 4
1.2 Organization .............................................. 5

2 Background

2.1 Xen Architecture ............................................ 6
2.2 Scheduling ................................................. 8
   2.2.1 Scheduling Aside: Virtual CPUs and Context Switches .... 11
   2.2.2 Fundamental Scheduling Problem ....................... 12

3 The Worldswitch Mechanism

3.1 Worldswitch Design ........................................ 14
3.2 Overview ................................................. 15
3.3 Authentication Via the Worldswitch Table ................... 17
3.4 Requesting Worldswitch Entries ............................ 18
3.5 Tracking Outstanding Network Requests .................... 19
3.6 Preventing Worldswitches ................................... 21
3.7 Context Switching without Involving the Scheduler ........ 22
3.8 Execution in Driver Domain .............................. 23
3.9 Switching Back ........................................... 24
3.10 Adherence to Xen Philosophy ........................................... 24

4 Evaluation ................................................................. 25
  4.1 Experimental System ................................................ 25
  4.2 Bandwidth Performance ........................................... 26
  4.3 CPU Usage .......................................................... 27
  4.4 Single Core vs Multicore ........................................ 33
  4.5 Latency .............................................................. 39
  4.6 Conclusions .......................................................... 45

5 Related Work ............................................................ 46

6 Conclusions .............................................................. 62
  6.1 Summary ............................................................ 62
  6.2 Future Work ........................................................ 63

Bibliography .............................................................. 66
Illustrations

2.1 Xen architecture showing driver domain and unprivileged guest domains

2.2 Xen split-end driver (this figure is reproduced from Chisnall [12]).
The shared memory ring is shown split across two domains. The light
gray part of the ring shows the requests sent by the front end in the
unprivileged domain while the darker segment of the ring shows the
response from the back end in the driver domain. The arrows
indicate the event channels that each domain uses to alert the other
domain to the presence of a request or a response on the memory ring.

3.1 A high-level overview of the steps involved in the worldswitch
mechanism ................................................................. 16

3.2 A guest requests an entry in the driver domain’s worldswitch table
and receives an authentication number in return ........................... 20

3.3 Guest 1, pinned on physical CPU 2, worldswitches to VCPU2 on
Domain 0 and at the same time prevents Guest 2, also pinned on
physical CPU 2, from worldswitching until the first worldswitch
completes ................................................................. 22

4.1 Aggregate bandwidth when streaming to all guests simultaneously 28

4.2 Aggregate bandwidth of transmitting guests .......................... 29

4.3 Average and individual bandwidth when streaming to all guests
simultaneously .......................................................... 30
4.4 Average and individual bandwidth when streaming from all guests simultaneously ........................................ 31
4.5 CPU usage for receive workloads ........................................ 33
4.6 CPU usage for transmit workloads ..................................... 34
4.7 Bandwidth for 1 guest receiving with different core configurations ........................................ 35
4.8 Bandwidth for 1 guest transmitting with different core configurations ........................................ 36
4.9 CPU usage for 1 guest receiving with different core configurations ........................................ 37
4.10 CPU usage of 1 guest transmitting with different core configurations ........................................ 38
4.11 Latencies pinging to a single guest under different core configurations ........................................ 40
4.12 Latencies pinging from a single guest under different core configurations ........................................ 41
4.13 Latency pinging to one guest while 11 other guests run I/O intensive workloads ........................................ 42
4.14 Latency pinging to one guest while 11 other guests run computationally intensive workloads ........................................ 43
4.15 Latency pinging to two guests while four guests receive streams and 4 run infinite loops. Two guests are left idle ........................................ 44
Chapter 1

Introduction

Virtual machines are useful for many things, including server consolidation and as sandboxes for research and testing. Virtualization is becoming an increasingly utilized and important part of computing infrastructure in many environments. For example, Google runs virtualized honeypots [29], Amazon farms out use of its EC2 cluster with virtual machines [1], and Sandia Labs [3] boasts one million Linux kernels running as virtual machines operating on only 4,480 physical machines. Fundamentally, running multiple operating systems on a single machine is a way to increase efficiency and versatility while conserving cost.

Virtualization, however, presents a challenge in that implementing a virtualized system in such a way that a guest operating system can achieve near-native performance is difficult. In particular, the issue of low-overhead virtual machine access to I/O devices still remains a significant hurdle. The reason for this is that while I/O devices are typically trusted parts of the overall system, guest operating systems are typically untrusted. All virtual machines running on a system must share the same physical hardware in a safe and fair way that allows each individual guest to behave as though it is a native operating system with full access to whatever types of devices it requires, regardless of the actual underlying hardware.
There are multiple approaches to virtualizing I/O. For hosted virtualization systems such as VirtualBox, KVM, and VMware Workstation, the I/O devices are inherently in the host operating system, as the device drivers are part of the host operating system. For virtualization systems implemented as bare-metal hypervisors, there is some variation in the location of the device drivers. VMware ESX supports I/O virtualization by placing the drivers in the hypervisor, thus granting the guests access to the I/O devices through a hypercall. Xen and Microsoft’s Hyper-V, on the other hand, follow the driver domain model of virtualization, and maintain the I/O devices in a separate, special, fully privileged (but protected and isolated) driver domain, while the guests remain unprivileged.

The hosted and VMware ESX solutions can provide I/O access at a much lower cost than the driver domain solution because a hypercall by a paravirtualized driver to access the device is cheaper than scheduling a different domain. The value to maintaining the driver in a separate domain is that unprotected access to the driver can in itself create a problem. Driver code is notoriously buggy, and in fact device drivers have been shown to contain a disproportionately high number of the bugs found in an operating system [13]. This situation can be made even worse when multiple different operating systems must have their own driver software emulated while accessing the underlying kernel software. In the worst case, the device driver code can crash the entire hypervisor. By maintaining protected access to the I/O drivers and providing a generic split-end driver for access, Xen offers a robust way of
allowing guests access to the device drivers but in the process suffers from reduced network performance.

This thesis presents a method of providing increased networking performance in Xen while still maintaining driver protection. I design and implement a mechanism by which guests may operate the driver domain directly in response to their outstanding network I/O requests without incurring a scheduling cost. Isolation is preserved because the guests only have a means to operate the driver domain, they are not gaining access to the drivers themselves.

Virtualized networking is a nontrivial problem and is a fundamentally harder problem than any other kind of I/O because of the unpredictable nature of network traffic. Network packets can arrive without being solicited, whereas for other typical types of I/O, if a request arrives unexpectedly, there is a more troubling issue than I/O scheduling. Furthermore, in the protected domain model of Xen, there is an unavoidable inefficiency in scheduling. A guest domain runs until either it has used up its allotted scheduler time, it is preempted by a higher priority domain, or it voluntarily yields the processor. In any of these cases, a new domain is scheduled in accordance with whatever scheduler algorithm is being used. For the purposes of processor use, the privileged driver domain is treated as just another domain, and there are no concessions made for outstanding I/O requests. In other words, a guest may be blocking, waiting on receipt of a network packet, but the scheduler is no more likely to run the driver domain next than it is any other domain. In the worst case,
all of the guests could be waiting to use the I/O devices without the driver domain being any more likely to be scheduled.

The observation is that a guest that is doing nothing but blocking, waiting on the driver domain, is wasting time that could be more efficiently spent running the driver domain. The proposed solution is to allow the guests to use this wasted time to run the driver domain. There is a direct analogy to this solution in LRPC, the lightweight remote procedure call described by Bershad et al. [7]. An LRPC is a comunication facility for circumventing the scheduler, and it was designed and optimized for communication between protection domains on the same machine. LRPC is precisely parallel to the mechanism I wish to create in Xen, that is, I want a system for communication between the protected guest domains that does not involve the scheduler.

1.1 Contributions

In this thesis, I describe the worldswitch, a mechanism designed for communication between guest domains and the driver domain, such that a guest blocked and waiting on a response from the network may use its scheduled time to run the driver domain on its own behalf without waiting for the intervention of the scheduler. The contribution of this mechanism is that it provides a way to reduce the virtual machine I/O performance cost while preserving the principles of protection and isolation between computing domains. I do this by showing that it is possible to bypass the
scheduler for faster network access without compromising system safety or stability. I demonstrate that the worldswitch mechanism allows virtual machines to achieve higher individual and aggregate bandwidth, as well as lower latency, than is possible with unmodified Xen. Furthermore, with this mechanism, access to the network is still fairly distributed across all running guests. Finally, the worldswitch mechanism accomplishes this without any specialized hardware or major alterations to the Xen code, demonstrating that this mechanism is cheap in terms of software engineering effort and hardware implementation in addition to reducing the cost of network I/O virtualization.

1.2 Organization

This thesis is organized as follows. Chapter 2 provides relevant background information describing Xen's driver domain architecture with emphasis on scheduling and virtual CPU structures. Chapter 3 details the design and implementation of the worldswitch mechanism. In Chapter 4 I present and discuss the results of my evaluation of the worldswitch mechanism. Chapter 5 provides an overview of related work on remote procedure calls and network I/O virtualization, summarizing both hardware and software solutions, and emphasizing how the work in this thesis differs from or is synergistic with other work. Finally, in Chapter 6 I summarize my results and the effect of the worldswitch mechanism on the existing Xen solution.
Chapter 2

Background

This chapter discusses some basic mechanisms of Xen. This background information is necessary for understanding the design and implementation of the worldswitch mechanism presented in this thesis.

2.1 Xen Architecture

The philosophy of Xen as a virtualization system is to minimize the size of its trusted computing base. This is motivated by the fact that a smaller hypervisor is easier to review for correctness. As a result of this philosophy, Xen does not provide any device drivers or user interfaces; all it provides is a capsulated environment known as a domain in which guests run. One of these, the first to boot, is the fully privileged Domain 0, which contains a fully functional operating system with direct access to the hardware. All other domains are referred to as DomUs, indicating an unprivileged domain. It is possible to delegate some of the privileges and responsibilities of Domain 0 to a DomU guest, but by default, the only way to access the hardware devices is through Domain 0. Furthermore, Domain 0 provides the user interface to the hypervisor, the software platform on top of which both Domain 0 and the DomU guests operate. The domain scheduling mechanism is located in the hypervisor. The
Figure 2.1: Xen architecture showing driver domain and unprivileged guest domains

The hypervisor is also responsible for trapping any privileged instruction performed by a guest, executing it, and returning control to the unprivileged domain in much the same way as a kernel trap is implemented. The entire structure of hardware, hypervisor, Domain 0, and unprivileged domains is shown in Figure 2.1.

Each guest domain has its own virtual network interface which is connected to the physical network interface by means of a software Ethernet bridge located in the driver domain, which is typically Domain 0. Of particular significance is the way in which drivers are split across the driver domain and the guest domain. Xen employs a split driver model in which each driver is in two parts, with the front end in the guest domain and the back end in a driver domain, which can be either Domain 0 or some dedicated driver domain. (For purposes of simplicity in this work, I will assume throughout that the driver domain is Domain 0.) In addition, a shared memory ring buffer is used to send packets across the two halves. Communication is achieved by
means of this ring buffer and an asynchronous event channel. When a domain needs to use the driver, it pushes a request onto the ring buffer from its end and sends an interrupt, using the event channel, to the other domain. When the other domain is woken by the scheduler, it sees the interrupt on the event channel, retrieves the request from the ring buffer, processes it in whatever way is appropriate, and pushes a response onto the ring buffer and sends an interrupt over the event channel in return. This architecture is shown in Figure 2.2.

2.2 Scheduling

When a domain has an interrupt pending, the hypervisor will wake, or tickle the domain and the scheduler will reevaluate which domain should be running. If the domain with the pending interrupt has a higher priority than the currently running domain, there will be a reschedule and the higher priority domain will be allowed to preempt the currently running domain.

Xen ships with two schedulers and a variety of configuration options. Only the default configuration is described here, as that was used for all experiments. Many different scheduling configurations are possible and can produce mixed results, particularly with respect to I/O performance, as has been shown by Ongaro et al. [26], but in this thesis I consider only the default scheduler. I do not consider the Simple Earliest Deadline First scheduler because it has been largely abandoned in favor of the newer credit scheduler. The reason given for this is that the credit scheduler signifi-
Figure 2.2: Xen split-end driver (this figure is reproduced from Chisnall [12]). The shared memory ring is shown split across two domains. The light gray part of the ring shows the requests sent by the front end in the unprivileged domain while the darker segment of the ring shows the response from the back end in the driver domain. The arrows indicate the event channels that each domain uses to alert the other domain to the presence of a request or a response on the memory ring.
cantly improves scheduling on multiprocessor machines, making it the more suitable choice for modern machines [4]. As for the default configuration of the credit scheduler, the results of Ongaro et al. show that none of the different possible scheduling arrangements are sufficient to address the fundamental scheduling problem inherent to a driver domain model of virtualization. This problem will be discussed in more detail later in this chapter.

By default, Xen uses a credit scheduler, with tickling and boosting enabled. The credit scheduling configuration assigns each domain some number of credits (the default is for all domains to be created with an equal number of credits and to debit credits away from each domain at the scheduler interrupts that occur every 10ms). When the sum of all credits for all domains becomes negative, new credits are issued for all domains. However, the exact number of credits for a domain is irrelevant, as the scheduler determines priority in scheduling solely by whether a domain is in an OVER or an UNDER state, where OVER indicates that a domain has used up all its credits and UNDER indicates that it has credits remaining. Furthermore, the scheduler selecting a domain does not mean that the domain immediately runs; it means that the domain is inserted into the run queue, behind whatever other domains are already there, and the run queue operates in a first-in-first-out fashion relative to its peers in the same state. A domain at the head of the run queue will operate for three consecutive scheduling intervals, provided it has sufficient credits.
The tickling feature of the scheduler, as mentioned above, is the mechanism by which the hypervisor will force the scheduler to reevaluate which domain is running in the event of a pending interrupt. In this case, if the domain with the waiting interrupt is in a higher priority state than the currently running domain, the scheduler will immediately run the domain with the pending interrupt.

Additionally, the default configuration in Xen allows for a BOOST state. A domain is considered to be in BOOST state when it receives an interrupt while idling, provided it has credits remaining. A domain cannot be BOOSTED if it has exhausted all of its credits. The BOOST state gives a domain higher priority than other domains, so that with tickling enabled, it is more likely to be able to preempt the currently running domain when the scheduler performs its priority evaluation. Also, a domain may at any time voluntarily yield the processor and forfeit its scheduled interval by invoking a YIELD operation.

2.2.1 Scheduling Aside: Virtual CPUs and Context Switches

To say that the scheduler selects a domain is an abstraction, though a useful one. Scheduling works by scheduling virtual CPUs rather than by scheduling the domains themselves.

Every guest domain runs on one or more virtual CPUs (VCPUs), and each VCPU may either be pinned to a specific physical CPU or allowed to float between the physical CPUs as determined by their availability. The default configuration is for
Domain 0 to be instantiated with as many virtual CPUs as there are physical CPUs and every guest is instantiated with exactly one virtual CPU. Unless specified otherwise by the user, all VCPUs will float rather than being pinned. Every virtual CPU has an associated state, such as RUNNING, PAUSED, or BLOCKED.

The main function of the scheduler, then, is to save the context of a running CPU, block it from executing, and jump to the execution stack of the domain saved on the virtual VCPU that has been selected to be run next. The selection of the specific VCPU is specific to the particular scheduling algorithm being used; it is not part of the main scheduler function.

2.2.2 Fundamental Scheduling Problem

A driver domain model of virtualization suffers from a fundamental problem of scheduling in that the driver domain is scheduled in exactly the same way as the guests. The driver domain must be run in order to perform I/O work for guests, but there is no notion of scheduling the driver domain directly in response to the guest domains. The scheduling modifications of tickling and boosting were designed to address this problem by making it more likely that a domain with pending interrupts will be able to preempt a domain without pending interrupts. However, as has been noted previously, the work of Ongaro et al. [26] shows that these modifications are insufficient. Tickling and boosting can only make it more likely that the driver domain will be scheduled as needed, however there is still no guarantee. Even more
problematic is that the driver domain can run out of credits. All work done by the
driver domain is deducted from the credits of the driver domain even if all the driver
domain is doing is performing work on behalf of guests. Guests performing I/O inten-
sive operations may cause the driver domain to run out of credits while still retaining
credits of their own. If that happens, all guests performing I/O operations will be
blocked until all guests run out of credits. The worldswitch mechanism provides a
way for guests to circumvent this problem by performing driver domain functions re-
gardless of the scheduling situation. In addition, a guest that performs a worldswitch
will have all the work done by the driver domain on its own behalf debited from its
own credits rather than from the credits of the driver domain. This is not only a
more accurate accounting, it will improve scheduling for all guests on the system by
not using up the driver domain's credits when performing I/O work.
Chapter 3

The Worldswitch Mechanism

3.1 Worldswitch Design

The worldswitch mechanism allows a running guest to usurp an idle virtual CPU belonging to Domain 0 and use it to run the back end of the driver domain and fulfill outstanding network requests on its own behalf. Guests do this by relinquishing a physical CPU and using that to provide the physical hardware on which to operate a virtual CPU belonging to the driver domain. Typically, a guest would use this mechanism only if it has no other meaningful work to do and is blocked waiting on a response from the network. However, it is possible that a guest may prioritize I/O higher than background computation, and, accordingly, there is nothing that would prevent such a guest from worldswitching at any time. In my prototype implementation, I assume that all guests prioritize computation and I/O equally, and therefore guests only perform worldswitches when waiting for the network. Finally, the design of the worldswitch mechanism should preserve the principles of protection and isolation, thus the driver domain should have some means of controlling which guests are allowed to use which of its functions. In its turn, a guest should have a means of authenticating itself to the driver domain. In designing the worldswitch mecha-
nism, I used the Lightweight Remote Procedure Call of Bershad et al. [7] as a model. The work of Bershad et al. and its influence on the worldswitch mechanism will be discussed in Chapter 5 of this thesis.

3.2 Overview

At a high level, the worldswitch mechanism works as illustrated in Figure 3.1 and described below. In this description, although guests are discussed as performing operations, every operation that the guest performs is executed either from the hypervisor or the generic Xen front-end network driver interface. In no case was any guest OS altered to be made aware of the worldswitch mechanism. This is possible because, although I refer to a guest as executing an idle loop, what actually occurs is that an idle guest executes a hypercall such that the hypervisor’s DO_BLOCK code begins execution. This code is an infinite loop in which the hypervisor will periodically check for pending events and interrupts. If an interrupt occurs, the loop will break, and the hypercall will return control to the guest.

The operation of the worldswitch mechanism consists of the following steps:

- When the driver domain boots, it creates a worldswitch table in which to store authentication information about guests.

- When a guest boots, the guest requests an entry into the worldswitch table as part of the handshaking process that establishes communication between the front and back ends of the network drivers. The driver domain adds the
Figure 3.1: A high-level overview of the steps involved in the worldswitch mechanism requesting guest ID along with a function pointer and an authentication number.

The authentication number is returned to the guest.

- While a guest performs I/O operations, it keeps a publicly accessible bit flag indicating whether or not it has outstanding network requests.

- When a guest is blocked, if it has outstanding network requests and no other guest is preventing it from worldswitching, it is allowed to perform a worldswitch by calling a hypervisor function.
• The worldswitch function consults the worldswitch table, and if the requesting guest has supplied the correct authentication number it performs a context switch from the guest VCPU to one of the idle VCPUs assigned to Domain 0 and blocks all guests that would use the same idle VCPU from performing a worldswitch.

• In the driver domain, the function specified in the worldswitch table that corresponds to the requesting guest is executed. The function does not return, but calls a return worldswitch function.

• The return worldswitch function releases all the guests that were prevented from calling a worldswitch and performs a context switch back into the blocking loop of the calling guest’s virtual CPU.

The mechanisms by which these steps are performed are discussed in more detail in the succeeding sections.

### 3.3 Authentication Via the Worldswitch Table

The worldswitch table is a memory page reserved for the use of and access by the driver domain. This structure is used as a table in which to store information about which guest may context switch into which functions of the back end of the network driver. Each entry in the table consists of an index number which is also used as an authentication number, the ID of a domain, and a pointer to a function in the driver
domain. This table is created during the network setup process that occurs when the driver domain boots.

This table is implemented as a specialized version of Xen's preexisting grant table structure. Xen already provides a mechanism for creating shared memory pages with its grant table structure. The Xen grant table is a bare-bones shared memory page between a domain and the hypervisor. To create a worldswitch table I found it convenient to leverage this grant table structure.

This table is a general purpose and flexible solution that can be used to allow any guest to call any specific function in the driver domain, or to give some guests special privileges that are denied to others. Although I focus here only on guests that all have equal need to operate the backend of the network, this table could be used as an authentication and bookkeeping method for a variety of other purposes.

For the purposes of evaluating this work, I did not take advantage of the flexibility provided by the worldswitch table. All guests switch into the same function and all guests are automatically granted the privilege of worldswitching.

### 3.4 Requesting Worldswitch Entries

Guests must explicitly make it known to the driver domain that they would like the ability to execute functions on their own behalf, and the driver domain will only add the guest to its worldswitch table in response to this explicit request. In addition, the guest must have some way of proving to the driver domain that it has been
granted this ability. As a guest boots and sets up the front end of the network driver, it makes a hypercall requesting an entry in the driver domain's worldswitch table. If the request is granted, an entry is added to the worldswitch table containing the requesting guest's ID and a pointer to the function in the driver domain that the driver domain will allow the guest to use. This entry is assigned a number which is used as an index number, and this number is returned to the guest as the return value of the hypercall. The process of creating worldswitch entries is illustrated in Figure 3.2. The guest will use the returned index to establish the right to perform a worldswitch and stores it as a field in one of the shared memory pages that Xen already provides and which is shared pair-wise between every guest and the hypervisor. The shared memory page was chosen for convenience.

3.5 Tracking Outstanding Network Requests

A guest will perform a worldswitch only when it has outstanding network requests. Since a running guest thus has to keep track of the state of its outstanding requests, I add a field in the shared memory page mentioned in the preceding section. The guest updates this field as appropriate and the hypervisor can simply look up this value when necessary. There is nothing preventing a guest from fraudulently claiming that it has pending requests when it does not. This is not a significant issue because a guest that did this would gain nothing. Guests only operate specific functions in the back of the network relating to their own pending requests. If no pending requests
Figure 3.2: A guest requests an entry in the driver domain's worldswitch table and receives an authentication number in return.

If the guests actually exist, the worldswitch will simply return to the guest. Since the guests only worldswitch while they are idle already, they will not be prevented from doing useful work by pointless worldswitches. At worst, a fraudulent guest may prevent guests that have actual network requests from worldswitching, but since a worldswitch without pending network requests would return quickly, this is not likely to be a significant concern.
3.6 Preventing Worldswitches

The driver domain has a limited number of idle VCPUs for guests to operate, and two guests may not both operate a driver domain VCPU at the same time. In order to ensure this I implement a system to prevent guests from performing worldswitches.

When a guest performs a worldswitch, the driver domain virtual CPU it is given to operate is assigned based on the ID of the CPU of the worldswitching guest. For example, in a quad core system, the driver domain will have four virtual CPUs, and the guests will be pinned onto cores 1-3. No guest should ever be pinned to core 0, as that core is reserved exclusively to the driver domain and one of its virtual CPUs. Say a guest pinned onto physical CPU 2 performs a worldswitch. It will be switching onto virtual CPU 2 of the driver domain. If there are any other guests pinned onto physical CPU 2, they should not perform a worldswitch at the same time as the first guest. This operation is illustrated in Figure 3.3. If this second worldswitch were to be allowed, two different guests would be attempting to context switch into the same virtual CPU and the system would become unstable.

The process of preventing such a second worldswitch is a matter of keeping track of appropriate bit fields, again using Xen's shared info page. Each guest has a field marking whether or not it is allowed to perform a worldswitch. When a guest makes a call to the worldswitch function, the hypervisor checks this field, and if the guest is not itself currently prevented from performing a worldswitch, the hypervisor will call a function that will fill in this field for all guests pinned to the same physical CPU.
Figure 3.3: Guest 1, pinned on physical CPU 2, worldswitches to VCPU2 on Domain 0 and at the same time prevents Guest 2, also pinned on physical CPU 2, from worldswitching until the first worldswitch completes as the worldswitching guest. If, instead, the requesting guest is currently prevented from worldswitching it continues idling in the hypervisor loop from which it made the request. When a guest returns from a worldswitch, the hypervisor will clear that bit field for all guests pinned to that same CPU. This will allow them to perform a worldswitch if they have the opportunity.

3.7 Context Switching without Involving the Scheduler

The point of the worldswitch mechanism is to be able to context switch from one domain to another without involving the scheduler. Xen normally executes context switches out of the main function of the scheduler, but a large part of that
context switch function involves internal bookkeeping and virtual CPU selection internal to the scheduler. By stripping out much of the pointer updates, it is possible to implement a minimalistic hand off function that does nothing more than update the necessary registers and stack pointers before executing a call to Xen's reset_stack_and_jump trampoline. The hand off function trampolines into a slightly modified IRQ-handling function in the driver domain, and from there the function specified by the worldswitch table can be called directly.

3.8 Execution in Driver Domain

In the backend of the driver domain, the function registered in the worldswitch table is executed. This function does not return, but instead executes a return worldswitch which will be described in the next section.

Though in my implementation I provide the flexibility for every guest to have a different policy on what function it calls and what work that function performs, in order to test and evaluate the system, all guests switch into the same function in the backend of the network driver, which calls the already utilized and implemented Xen functions to execute the pending network requests and push the result onto the shared ring buffer.
3.9 Switching Back

To return to the context of the switching guest, the driver domain issues a hypercall that executes another stripped down context switch function and trampolines back into the blocking loop of the worldswitching guest. In this blocking loop, the hypervisor already checks for the pending interrupts that have been left by the execution of the driver domain.

3.10 Adherence to Xen Philosophy

In proposing the worldswitch mechanism, I have been careful to adhere to Xen's philosophy of minimizing the trusted code base and keeping mechanisms as general as possible. Though I have implemented the worldswitch mechanism as a means of allowing a guest to run the back end of the network driver on its own behalf, at its heart, the worldswitch mechanism is only a means of context switching from a blocked guest into the driver domain without involving the scheduler. Because of the worldswitch table mechanism, I allow the driver domain to implement any policy of work that a guest may perform via the worldswitch. I have also attempted to minimize code additions by stripping down the already existing Xen mechanisms in order to implement a bare-metal worldswitch mechanism.
Chapter 4

Evaluation

This chapter evaluates the implementation of the worldswitch mechanism in two ways.

First, I test the bandwidth performance of the system under increasing numbers of guests running both transmit and receive workloads and examine the CPU utilizations to explain the results. Second, I investigate the latency of the system under a variety of workloads with both the stock configuration and with the worldswitch mechanism enabled. I also show how the worldswitch mechanism performs on a single core system as compared to a multicore system. The goals in this evaluation are to demonstrate both improved network performance and a solution to some of Xen's scheduling problems.

4.1 Experimental System

I implemented the worldswitch mechanism in Xen3.5 unstable (changeset 20425; November 11, 2009) running on a 3GHz AMD engineering sample comparable to a Phenom II quad core processor, with 4GB of RAM and a 10 gigabit Ethernet interface. Under native Linux, this system can send and receive at line rate. Unmodified, the Domain 0 kernel can receive at approximately 6.2 Gbps and transmit at
approximately 7.7 Gbps. When sending to and from one guest, receive and transmit performance drops to around 1.9 Gbps and 5.8 Gbps, respectively.

For my tests, Domain 0 was booted with four virtual CPUs and all guests were pinned onto physical cores 1-3, with 0 being reserved for Dom0's VCPUs. All guests were identical. Each one was provisioned with a read-only file system and a read-write "/var" filesystem and was booted with exactly one virtual CPU. All guests had the same scheduling priority. An end system was used for streaming and ping tests. For the end system, I used a 3GHz AMD Athlon 64 X2 Dual Core Processor with a 10 gigabit Ethernet interface running native Linux. I verified that this end system can transmit and receive at line rate, and in no case was the end system a bottleneck for any tests.

4.2 Bandwidth Performance

As a simple test of the efficacy of the worldswitch mechanism, I used the end system to stream TCP packets to and from all guests simultaneously using netperf [2]. Streaming tests use all possible network resources in a guest. I perform these transmit and receive tests while increasing the number of guests on the system, starting with 1 and ending at 21. For simplicity, after 3 guests I show guests incremented by 3. I chose this increment so that each core has the same number of guests pinned to it. The results of the worldswitched system under these tests are compared to the same tests with worldswitching deactivated. The results of the receive workload are
shown in Figure 4.1, and the results of the transmit workload are shown in Figure 4.2. These graphs show the aggregate bandwidth plotted against the number of guests running. Under both workloads the worldswitched guests achieve a higher aggregate bandwidth than under the stock Xen configuration. On average, under the receive workload, this improvement is around 200 Mbps while under the transmit workload, the improvement is about 500 Mbps. In both cases, the worldswitch achieves an average of about 10% higher bandwidth than stock Xen. In the receive test, the aggregate bandwidth peaks at 3 guests and then declines as guests are added; these results are discussed further in Section 4.3.

The individual bandwidth for each guest, plotted against the number of guest in these tests, is shown in Figures 4.3 and 4.4. Guests in the stock Xen configuration are shown as circles, whereas guests with the worldswitch implemented are marked with an x. Average bandwidth across the system is indicated with a thin horizontal line for worldswitched guests and a shorter and thicker line for unmodified guests. For both configurations, average bandwidth decreases as the number of guests increases, which is an expected result. In addition, overall the bandwidth is distributed fairly evenly across all guests.

4.3 CPU Usage

The CPU utilization for the experiments in the previous section is shown in Figures 4.5 and 4.6. In these graphs CPU utilization for Domain 0 is shown by the lines at
Figure 4.1: Aggregate bandwidth when streaming to all guests simultaneously
Figure 4.2: Aggregate bandwidth of transmitting guests
Figure 4.3: Average and individual bandwidth when streaming to all guests simultaneously.
Figure 4.4: Average and individual bandwidth when streaming from all guests simultaneously
the top, with the dotted line showing the stock configuration and the dashed line showing the worldswitched configuration. The other lines show the average usage across the three cores running the guest domains. The dash-dot line indicates the stock configuration, whereas the solid line indicates the worldswitched configuration. The x-axis shows increasing number of guests on the system, as in the previous graphs.

As shown in Figure 4.1, the bandwidth performance under a receive workload peaks at 3 guests; also as shown in Figure 4.5, at three guests the usage of Domain 0 reaches 100%. In addition, under the worldswitched configuration, the CPU usage by Domain 0 is lower and the usage by the guest domains tends to be higher than under the stock configuration. This is because a guest performing a worldswitch is providing extra parallelism to Domain 0 with its physical core, but this extra time is credited to the guest rather than to Domain 0. This extra parallelism also explains why the worldswitched configuration can achieve higher performance than the stock configuration even once Domain 0 reaches 100% utilization.

Under the transmit workload, neither Domain 0 nor the guest cores reach maximum CPU utilization. Usage increases as guests are added. This is consistent with the observation that under a transmit workload, the bandwidth does not fall off as the number of guests increases in the way that bandwidth does under a receive workload. A transmit workload is not nearly as processor intensive as a receive workload (this is also why the guests transmit at a higher rate than they can receive).
Finally, Domain 0's CPU utilization reaches slightly above 100%. This is due, however, to measurement inaccuracy in Xen's provided xm top tool.

4.4 Single Core vs Multicore

I also investigated the performance of the worldswitch mechanism with the system restricted to one core. For this experiment I ran one guest on the system and ran both the transmit and receive workloads. Figures 4.7 and 4.8 show these results plotted against the bandwidth performance of a single guest on the system with
Figure 4.6: CPU usage for transmit workloads
Figure 4.7: Bandwidth for 1 guest receiving with different core configurations

all four cores enabled. The bandwidth for guests under the stock configuration is shown as a circle while the bandwidth for guests implementing the worldswitch is marked with an x. Under a receive workload, though the worldswitch mechanism under 4 cores shows substantially higher bandwidth performance, under a single core there is virtually no difference. Actually, the worldswitch mechanism causes a slight, insignificant reduction in performance (2 Mbps). However, under a transmit workload, the worldswitch mechanism causes an increase in bandwidth performance both for a quad core and a single core system.
Figure 4.8: Bandwidth for 1 guest transmitting with different core configurations
Figure 4.9: CPU usage for 1 guest receiving with different core configurations

In order to explain the bandwidth differences, Figures 4.9 and 4.10 show the CPU utilization for the above results. The CPU usage by Domain 0 under a worldswitch configuration is indicated by a square whereas the utilization under a worldswitch implementation is marked with a triangle. Under the receive workload, the worldswitch mechanism causes a reduction in the usage of Domain 0 and an increase in the usage of the running guest. Under a single core, this makes no difference. There is only 1 core and it is already at 100% utilization. Under the transmit workload, however, the usage of Domain 0 is not a bottleneck the way it is under a receive workload.
Figure 4.10: CPU usage of 1 guest transmitting with different core configurations
4.5 Latency

I ping to and from guests to measure the response latency. As in the preceding section, this is done using a single guest, both with four cores and with the system restricted to one core. The results for these tests are shown in Figures 4.11 and 4.12. Average ping latency is indicated by a triangle, and the error bars show the standard deviation. When pinging to a single guest, the worldswitch mechanism reduces the average latency by about .3 milliseconds on a quad core and .1 milliseconds on a single core system. When pinging from a guest, the worldswitch mechanism reduces the latency by 0.02 milliseconds on a quad core and 0.05 milliseconds on a single core system.

That the worldswitch mechanism makes much less of a difference when pinging from a guest is consistent with earlier results. Transmitting is a much less demanding workload, and particularly when running a single guest on a quad core system, there is already parallelism present and not very much need for a worldswitch. However, that latency is lower in all configurations with the worldswitch mechanism enabled indicates that the worldswitch successfully addresses some of the scheduling problems inherent to Xen.

In order to further investigate the effects of the worldswitch mechanism on the latency of the system, I booted twelve guests and configured them with three different workloads. The first test concentrates on I/O intensive workloads. This test sends pings to the first guest while the other eleven receive TCP streams. The results from
Figure 4.11: Latencies pinging to a single guest under different core configurations
Figure 4.12: Latencies pinging from a single guest under different core configurations
Figure 4.13: Latency pinging to one guest while 11 other guests run I/O intensive workloads

these tests are shown in Figure 4.13. The TCP receive performance of each guest is plotted as an x or a circle, for worldswitched and stock guests, respectively, while the ping latencies are represented as triangles. The guests running worldswitch show a 0.1 ms reduction from the guests running stock Xen. The standard deviation on both configurations is almost identical.

Again running twelve guests and sending pings to the first guest, I altered the workload on the system to be computationally intensive by having each guest run
Figure 4.14: Latency pinging to one guest while 11 other guests run computationally intensive workloads

infinite loops to consume all available processing resources. The results from these tests are shown in Figure 4.14. The CPU time of each guest is plotted as a square. Here I see the worldswitched guests achieve a 0.2 reduction in latency from stock Xen. The standard deviation under both configurations is too small for the plotted error bars to be visible.

Finally, I mixed I/O and computational workloads and streamed to four guests while four guests ran infinite loops. Of the remaining four guests, I sent pings to
Figure 4.15: Latency pinging to two guests while four guests receive streams and 4 run infinite loops. Two guests are left idle.

two and left the others idle. These results are shown in Figure 4.15. This is the workload under which the benefit of the worldswitch in reducing latency is the most clear. Guests running stock Xen show ping latencies of 0.48 ms and 2.6 ms, while guests implementing the worldswitch show only 0.17 ms and 0.16 ms latencies.
4.6 Conclusions

This chapter has shown that the worldswitch mechanism can effectively address a problem with Xen's network I/O virtualization without altering the fundamental principles of Xen. The worldswitch mechanism is opportunistic—guests only perform worldswitch operations when they are blocked waiting on the network—thus, having worldswitching enabled does not alter the general behavior of the system. Instead, the worldswitch mechanism allows guests to use the network more efficiently. This is shown both in the higher aggregate bandwidth achieved by guests with worldswitching enabled, and in the reduced latency under various workloads.

These results show that the worldswitch mechanism is as scalable as unmodified Xen, in that the system remains stable running a large number of guests, all attempting to worldswitch at once. I was able to easily run 21 unprivileged guests on this hardware, each having a 2 GB file system and each running a benchmark server. From previous experiments on this test server, I have shown that under unmodified Xen this is the limit of guests that are large enough to do useful work that can be run on this hardware without overwhelming the processor due to memory constraints.
Chapter 5

Related Work

There has been a great deal of work done on network I/O virtualization performance. Rixner [32] discusses the growing need for efficient network virtualization, as the popularity of virtualization surges, and gives an overview of some of the different techniques for providing virtual machines with access to a physical network device along with some of the advantages and drawbacks for each. There is the Private I/O device system, employed in the first widely available virtualization platforms, the IBM System/360 and System/370. In this virtualization model, each guest has a physically distinct network interface, a solution that yields very high I/O performance. Gum [15] in fact claims that responsiveness to I/O requests are made so quickly that “any discrepancy with native execution should have no significant effect.” Parmelee et al. [28] describe in detail the mechanism by which guests are connected to each physical interface. The host system maintained detailed descriptions of every virtual machine’s I/O structure, with each table containing the existence and status of every virtual I/O element and its corresponding physical hardware. When a guest issued an I/O instruction, the host looked up the I/O address in its table to determine if the address was valid and if the I/O device was currently free. This private I/O solution did yield near-native performance, and is still employed today by the Power4 LPAR,
but it does not scale well due to the necessity of having distinct physical hardware for every guest. For this reason, shared I/O devices appeared as a solution for physically separated virtual machines. Under this model of virtualization, a virtualized spool-file interface was maintained by a specialized virtual machine called the I/O domain. All machines hosted on the system could read and write to virtualized spool files, and the hypervisor could interpret these reads and writes to determine whether the spool location was located on the local machine or a remote machine. If the location was to a remote machine, it would transfer control to the I/O domain. MacKinnon [20] details one such I/O domain, called the Remote Spooling Communications Subsystem (RSCS) on the VM/370. RSCS was a special-purpose operating system that could operate only as a virtual machine and was the means for a virtual machine to transmit data outside both itself and the real machine. This is the logical ancestor to the driver domain that this thesis is concerned with improving. Rixner [32] finishes his discussion with an overview of hardware solutions, focusing on the clever use of multiqueue NICs and Concurrent Direct Network Access, both of which will be covered in more detail below.

There has also been a great deal of work on remote procedure calls (RPC), of which the worldswitch mechanism is an example. Tay and Ananda [34] discuss the early history of RPCs. Though fundamentally, the RPC is a form of inter-process communication it began as a mechanism used exclusively for distributed systems. Thus, when Tay and Ananda discuss semantics of client and servers, they are considering
only client and server processes on different machines. They further define semantics of blocking and non-blocking RPCs, with blocking RPCs being those in which the client issues an RPC and waits for the response from the server before returning. A non-blocking RPC is one which returns to the client application immediately and somehow retrieves the results from the server at a later time. The worldswitch mechanism was designed as a blocking RPC. Tay and Ananda then discuss the Xerox Cedar RPC as one of the earliest implementations of remote procedure calls. In their seminal paper on this implementation, Birrell and Nelson [9] discuss the attractions of remote procedure calls and address a serious shortcoming in the previous RPC proposal found in Nelson’s doctoral dissertation [25], namely, semantics. In particular, Birrell and Nelson proposed precise semantics for calls in the face of machine and communication failures, binding, and suitable protocols for transfer of data and control between client and server. Also, Birrell and Nelson developed semantics that allowed pointers to be passed as arguments to RPCs. Finally, they proposed stubs, by which a remote call can be executed by means of a perfectly normal local call that then initiates the corresponding remote call in the stub procedure. Likewise, the worldswitch mechanism is invoked with a perfectly normal hypercall which invokes a specialized context switch function in the hypervisor.

Tay and Ananda [34] also discuss the Sun ONC/RPC, which at the time was the simplest of the RPC implementations, and could be easily ported to different architectures and systems. The Network File System (NFS) was originally built on
UDP-based RPC. As has been shown by Cattaneo and Persiano [11], the RPCs of NFS are easily and transparently extensible to support security operations. This makes it possible for a user to access an encrypted file system without needing to perform special operations. The worldswitch mechanism was implemented without considering security operations, but Xen does support encrypted filesystems, and since the worldswitch functions are located in the hypervisor, the worldswitch could be encrypted in the same way that Xen normally would encrypt hypercalls. Tay and Ananda continue their survey of RPC implementations by discussing the object-oriented Apollo NCA/RPC, designed on top of connectionless-oriented transport protocols. The NCA/RPC provided a wide variety of RPC calls, including extra routines for ping and quit packets, and call-back mechanisms that made it reliable and dependable in the face of network partitions or server crashes. However, because the RPC was so tightly integrated with components defined in the NPC, coding and porting was difficult. In contrast, the worldswitch implementation relies only on mechanisms that already exist in Xen, making it as portable as Xen itself. The Cambridge Mayflower Project RPC continued to introduce coding difficulties by implementing RPCs in a way that was not transparent. The entire Mayflower project was developed in order to “provide language-level support for the development of distributed computing systems [6].” It is interesting to note that the Mayflower supervisor kernel provided support for groups of lightweight processes running in environments referred to as
domains, to which system resources were allocated. These domains are the logical ancestor to the domains that Xen implements.

Tay and Ananda [34] then discuss the RPCs associated with two other large-scale architectural projects, The MIT Athena and the Modula/V from Stanford. The Athena was developed for educational purposes and the Athena RPC was a prototype implemented to evaluate design and applicability of an RFC under various constraints. The Athena RPC has no guarantee of delivery, and neither errors nor results are returned. Furthermore, the Athena RPC was implemented using two Unix processes and ports and suffers a performance and resource penalty. The Modula/V RPC, on the other hand, was very fast and very reliable. It was implemented with a team of processes to handle any client request. The server was called the dispatcher process and signaled workers in its team to handle any request that might cause the server to block. This multithreaded approach did provide very fast response time, however, the system suffered from an inefficiency of the V system. The V process provided no way for worker processes to reply directly to the original caller on behalf of the dispatcher process so worker processes had to execute the extra step of returning to the dispatcher process. The worldswitch mechanism does build indirectly on this parallelized approach, as the Athena example clearly shows the problems of using a simpler method. Even though the worldswitch is a blocking RPC, it executes using a physical core from the guest domain so that any other operations of the driver domain are not affected. Tay and Ananda complete their survey of RPCs by
mentioning the Rajdoot RPC. Rajdoot was most notable for incorporating measures for orphan detection and killing, however it implemented a number of mechanisms for dealing with reliability issues and also supported call-nesting capability [27]. This call-nesting makes Rajdoot the logical ancestor to Xen's multicall support in its hypercall implementation.

There have been a number of RPC implementations developed after Tay and Ananda's survey paper. There is the work of Hutchinson et al. [17] in implementing an RPC for the z-kernel, a configurable system kernel designed to simplify network protocol implementations. Hutchinson et al. proposed a novel technique for the design of an z-kernel RPC with virtual protocols. A virtual protocol is a header-less protocol that accepts messages from high level protocols and multiplexes them into lower-level protocols. By implementing an RPC on top of a virtual protocol rather than on top of IP or Ethernet, the RPC can support semantics for both without incurring the overhead of inserting an additional protocol between RPC and the network interface. This allows for simplicity and flexibility in implementing distributed systems over various network protocols. It is also a logical ancestor to the hypercalls that access Xen's generic device drivers.

There have been other notable attempts to optimize remote procedure calls. Johnson and Zwaenepoel [18] developed the Peregrine system specifically to improve the performance of network RPC calls. One of the ideas driving Peregrine is to offload as much responsibility as possible from the server to the client by forcing any argument
representation conversion to be done by the client. Furthermore, the server does not save any thread-specific state, as Johnson and Zwaenepoel note that saving neither register nor stack contents for server threads is necessary once the server thread has stopped executing. In designing the worldswitch mechanism, I attempt to follow the Peregrine principle of offloading as much responsibility as possible from the driver domain.

The Concert/C distributed programming model of J.S. Auerbach et al. [5] presents a flexible interface for RPCs. Concert consists of an endpoint modifier, which is the programmers’ specifications, and a network “contract” as a transport protocol. The goal of Concert is to provide a “minimal contract” between the network contract and the endpoint modifier. By separating the network semantics from the language semantics, Concert minimized the difficulties of mapping RPCs to different languages. Xen already provides a very generic hypercall system that, when coupled with its generic driver interface, allows me to build a similarly flexible and separated RPC for the worldswitch system.

Draves [14] describes an RPC implementation on the Mach 3.0 that took advantage of the hand-off scheduling optimization to significantly reduce the cost of an RPC. Black [10] describes the hand-off mechanism in the Mach as a means of taking a fast path through the scheduler. A thread “hands off” a processor to a different thread in a way that allows that different thread to be scheduled in a fashion that avoids many of the internal mechanisms of the scheduler. In particular, the hand-off
mechanism allows the newly scheduled thread to avoid waiting in the run-queue. Xen provides no such hand-off or fast-path scheduling mechanisms, and, in fact, one of the problems that Ongaro et al. [26] point out is that the scheduler in Xen merely inserts domains into a first-in-first-out run queue. The worldswitch mechanism completely bypasses Xen's scheduler, but it is conceivable that even with the worldswitch implemented, Xen's scheduler could be improved with the addition of some type of hand-off optimization.

Another optimization to RPCs has been presented by the work of Bershad et al. [7] on Lightweight Remote Procedure Calls (LRPC). In developing the worldswitch mechanism, I was inspired by this work. Bershad et al. identified an architectural limitation on the performance of procedure calls between protection domains in an unvirtualized system. When processes communicate across different protection domains in a typical remote procedure call, they must invoke a kernel call, and a processor's virtual memory context must be reallocated from one domain to the other via a scheduler operation that blocks a client's process and selects the necessary server process. There is significant overhead associated with this. Bershad implemented a lightweight remote procedure call designed to communicate directly between two protected domains via a kernel trap that bypasses the scheduler. This is directly analogous to the worldswitch mechanism that I propose here as a means of communicating between two different guest domains without involving the scheduler but maintaining the principle of domain isolation and protection.
Of special significance in LRPC is its leveraging of multiprocessor systems. Bershad et al. note that much of the context switch overhead is due to having to update the virtual memory registers of the hardware. LRPC takes advantage of a multiprocessor system by caching domains on idle processors and adding a kernel operation to check for an idling processor that is caching the domain needed for a context switch. If one is found, the procedure may be directly executed on the idle processor without a context switch and incurring only the overhead required to spin up the processor.

In a later paper on interprocess communication for shared memory multiprocessors, Bershad et al. [8] expand upon this idea of leveraging multiprocessor systems not just by opportunistically using any idle processor that might be available at the proper moment, pending the vagaries of the scheduler, but by actively ensuring that there will always be a physical processor present available and active for both the procedure call and the reply.

The worldswitch mechanism leverages the idea taking advantage of idling processors, proposed by Bershad et al. [7] for their LRPC mechanism, in order to reserve idle cores for switching. By default, on a multicore CPU, Domain 0 will initialize as many VCPUs as actual cores in the system. By keeping all but one of those VCPUs idle and pinning each guest to a specific core, a ready-made supply of available virtual CPUs exists for the guests to operate with their own physical core when they need to perform I/O operations. I have shown with the performance results of implementing the worldswitch mechanism that keeping these VCPUs idle except when operated by
the guest domains does not negatively impact I/O performance. I believe that providing an implementation specific to multicore systems is necessary and sufficient as a general proof of concept, in that any practical implementation of Xen will be realized on a multicore system. However, nothing would prevent the worldswitch mechanism from being implemented on a single core system, since fundamentally the mechanism is a low-level context switch between virtual CPUs, and the only constraint is that there be an idle VCPU available for a guest to operate.

My work on the worldswitch mechanism is more than a reimplementation of the work of Bershad et al., as the Xen environment presents unique challenges that lead me to expand on their work in the following ways. First and most importantly, unlike the environment which Bershad et al.'s work addresses, Xen provided no existing abstraction of or overhead-laden mechanism for scheduling the driver domain in response to requests from a guest domain. At most, the scheduler will tickle the driver domain in response to pending interrupts, but the driver domain may or may not have sufficiently high priority to be scheduled. Furthermore, there is no way to force the driver domain in Xen, even when scheduled, to immediately process the interrupt from the most recent requesting guest, as there may be other, higher priority work for the driver domain to perform. My worldswitch work does provide such a mechanism for operating the driver domain immediately on request of a guest domain. Second, LRPC is a general purpose mechanism for any domain to communicate with any other domain in the same machine. The assumption behind LRPC is that commu-
nication proceeds in a many-to-many fashion, whereas the worldswitch must address exclusively a many-to-one situation with one domain acting as a bottleneck resource. This led to implementation challenges of system instability resulting in the design of the previously described prevention mechanism to address the case in which multiple guest domains may attempt to operate the same Domain 0 VCPU.

Gupta et al. [16] summarize the reasons for the desireability of enforcing the principle of isolation in Xen. They note that encapsulated domains exist not only for fault isolation, but for performance isolation. Virtual machines are quite often instantiated with certain performance and service guarantees. Amazon’s EC2, for example, charges for space according to those principles. However, since the time spent running the driver domain on behalf of a virtual machine is credited to the CPU usage of the driver domain rather than to the requesting virtual machine, these performance guarantees are routinely violated. Gupta et al. propose a number of mechanisms for more accurate performance monitoring and accounting, as well as a control mechanism for enforcing performance guarantees based on these new tools. The worldswitch design partially addresses the same problems in that by running the driver domain on its own behalf, a guest will have the time it spends on network I/O credited to itself. Combined with Gupta et al.’s control mechanism, the worldswitch has the potential to increase the performance of running guests while more accurately enforcing fair scheduling.
I have already mentioned the work of Ongaro et al. [26] and his investigation of the performance of Xen's schedulers under varying workloads. For my latency tests, I leverage his experimental structure for representative workloads that demonstrate the efficacy of the worldswitch implementation.

The works of Ram et al. [31] and Santos et al. [33] are also synergistic to this thesis. Ram et al. note that a major source of overhead in Xen's driver domain model of virtualization is due to packet copying and demultiplexing. Traditionally in Xen, all packets are placed into the receive buffer in the driver domain. The driver domain must not only demultiplex the incoming packets, it must then copy the packets into the receive buffer of the guests for whom the packets are destined. This means that the driver domain must be able to access protected guest memory. This is achieved by Xen's grant mechanism, which has been mentioned in the context of this work as the parallel for the authentication method of the worldswitch mechanism, that is, the worldswitch table. The grant mechanism has its own associated overhead. In particular, grants are issued and revoked with each I/O operation. Ram et al. proposes a means of reusing existing grants in order to minimize the number of grant operations that must be performed. Additionally, they propose an I/O translation table that tracks all existing memory grants and so, when a shared memory page is needed, a domain may check for and use an already existing page in the I/O table rather than requesting a new grant for every operation. Santos et al.'s work describes the use of multiqueue NICs in a virtualization-specific way, such that each guest has
its own dedicated interface queue and packet demultiplexing is moved out of the driver domain and onto the NIC. Furthermore, a packet copy is eliminated, as the NIC places packets directly into guest memory without the need to copy them over the Ethernet bridge. Thus in this case, no grant needs to be issued at all, though the driver domain must still ensure that the guest buffers exist, but this may be done by consulting Ram et al.'s I/O translation table. These works, together with the worldswitch mechanism, complement one another as software solutions to eliminating performance barriers in Xen's networking implementation.

The worldswitch mechanism and the previously described complementary work are software solutions, but there have been hardware solutions proposed as well. Willman et al. [35] developed the Concurrent Direct Network Access (CDNA) solution for improving network I/O in Xen by presenting each guest with direct access to the network interface. This is done with an architecture that supports multiple contexts in hardware such that each context is presented to a guest as if it was a unique physical interface, and the ownership of each context is given to a guest rather than to a privileged driver domain. The hypervisor continues to guarantee memory protection and deliver interrupts, but the system is no longer limited by the same scheduling restrictions on the driver domain that the worldswitch was designed to address.

The Self-Virtualized Network Interface, or SV-NIC [30], proposed by Raj and Schwan, is a similar solution to CDNA in that Raj and Schwan are offloading some of the tasks of network I/O onto the virtual device. The SV-NIC leverages the periph-
erals found on high-end NICs to provide a virtual interface connecting the physical device directly to the guest. The hypervisor is still involved in configuring the virtual interface, but once configuration is complete, the guest can carry out most network tasks without hypervisor involvement. A key task for the self-virtualized device is to multiplex and demultiplex multiple virtual network interfaces in a single physical device. This is the same problem that Santos et al. approached with multiqueue NICs, but the Raj and Schwan solution creates one micro-engine context per virtual interface, and stores these in a pool of shared contexts belonging to a single micro-engine. The contexts are scheduled in a round-robin fashion.

Though CDNA and SV-NIC both rely on specialized hardware, McAuley and Neugebauer [21] argue that general purpose computer architectures should encapsulate all I/O systems in Virtual Channel Processors that would behave similarly to an embedded I/O processor. They point to the increasing tendency to offload I/O processing onto NICs, but unlike Raj and Schwan and Willman et al., they do not necessarily think this is a good idea, and instead propose a method of virtualizing the entire I/O system in a way similar to channel processors in mainframe computers. This provides both protection and separate scheduling domains, as well as a clean slate system that would give developers a general way to simply pick whether they wished to have their I/O performed on the main processor or offloaded to a NIC. This is not, in itself, a solution to the problems of virtualization I/O performance, but if implemented, it could provide more flexibility in dealing with those problems.
Even when no changes are made to the way that the guests interact with the hardware, it has been found that significant performance gains can be made simply by optimizing the software interface. Xen's provided virtual interface was designed to be very bare-bones in order to support a wide variety of guest operating systems, but at the same time this prevents the guests from taking advantage of many capabilities of the physical NIC. For example, Xen 2.0 offered no software support for a NIC's offload capabilities, and by adding checksum offload, scatter/gather DMA support, and TCP segmentation offload Menon et al. [22] were able to achieve virtualized network performance within 12% of native Linux.

The reason Menon et al. were so successful in optimizing Xen for performance was that their previous work was in developing the Xenoprof tool, which is a statistical profiling toolkit similar to Oprofile and implemented for a Xen environment. Menon et al. demonstrated the use of this tool in diagnosing performance overheads in Xen, focusing specifically on TCP send and receive workloads. In their succeeding paper [23], discussed previously, they were able to fix many problems that they found.

Another alternative solution to the performance barriers to networking in virtualized systems is to attempt to offload some of the driver work from the driver domain into the hypervisor. Menon et al. [24] attempted this with the TwinDrivers system, in which performance-critical send and receive operations of the network drivers are executed in the Xen hypervisor, while a second instance of the driver executes as usual split across the front and back of the network, with a single instance of the driver
data residing in the guest. Communication between the two instances is achieved by means of an upcall mechanism. This arrangement avoids the necessity of implementing an entire driver in the hypervisor, which is costly in terms of engineering effort, even more so given the mechanisms that had to be implemented in order to protect the hypervisor from notoriously buggy device driver code. Under the TwinDrivers system, all memory accesses from the hypervisor driver instance to the driver domain address space must pass through the software virtual machine mechanism, undergoing translation and checks, in order to ensure safe memory accesses. The worldswitch solution, in contrast, maintains complete performance isolation in Xen with a minimum of software engineering effort.

Finally, the idea of guests bypassing the scheduler in software in order to run the drivers on their own behalf has been proposed before [19] using the VMM-Bypass system, in which a specialized device driver called a guest module is implemented in each guest while a backend module is implemented in the hypervisor. The guest module creates virtual access points in the guest OS and accesses the backend module via a device channel. The backend module has direct access to the physical device, and there are access checks to maintain the principles of system integrity. For the worldswitch system, I instead follow the Xen principle of less is more and avoid the complex software engineering problem of designing what is essentially another split-end driver that is both more OS-specific and more privileged.
Chapter 6

Conclusions

6.1 Summary

Network I/O virtualization remains a barrier to efficient network performance in a driver domain model of virtualization. This is partially due to a fundamental problem created when the I/O device drivers are in a domain of their own; the scheduling policies cannot always provide enough runtime to the driver domain to expedite the I/O requests pending from and for the guest domains. In order to get around this problem, many virtualization solutions have hosted the device drivers in the hypervisor instead of a separate protection domain. However, I believe that the protection and isolation provided by the driver domain model of virtualization provides valuable advantages. In this thesis, I have proposed the worldswitch mechanism as a means of circumventing the scheduler in order to increase network performance while at the same time maintaining protection and isolation.

The worldswitch mechanism provides a guest that is blocked without any meaningful work and is waiting on outstanding network requests a means of using its physical processor to operate a virtual processor belonging to the driver domain in order to process its own network requests. This thesis has shown that this mechanism can
increase the aggregate bandwidth of the system by an average of 10%. Furthermore, this mechanism reduces the latency of running guests, indicating that the mechanism effectively addresses the scheduling problem.

Though the implementation presented in this thesis is specific to allowing a guest to process its own network requests, this mechanism provides a general framework for a guest to donate its physical CPU to run a virtual CPU of any other domain in order to perform work for itself directly and immediately.

This thesis has shown that the worldswitch mechanism has the potential to reduce the negative effects of the scheduling policy on the I/O performance of a virtualized system by providing a means of bypassing the scheduler without in any way fundamentally changing the nature of the virtualized environment. Additionally, the worldswitch mechanism does not involve the significant software engineering effort of developing specialized drivers or rely on specialized hardware and can be implemented in a way that conforms to the Xen philosophy of minimizing the trusted code base.

6.2 Future Work

Future work should concentrate first on redesigning the mechanism for preventing a second worldswitch to a VCPU if the first is interrupted by the scheduler and the means by which guests track outstanding network requests. The current methods, though functional, are extremely naive. For the prevention mechanism, currently I simply mark some guests as ineligible for switching. I would like to more intelligently
implement the worldswitch mechanism so that if execution in the driver domain is interrupted by the scheduler, the system cleanly pauses and saves that guest’s context on the driver domain’s VCPU so that other guests pinned to the same CPU can perform worldswitches.

Outstanding network requests are currently tracked by a simple bit flag, and all outstanding network requests are treated equally. This, combined with the naive prevention mechanism means that a guest sending pings has the same priority to perform a worldswitch as a guest sending large files, and whichever gets scheduled first is the one that gets higher performance while the other is marked as ineligible to perform a worldswitch. A more efficient method would be to only signal outstanding requests if there is some threshold number and/or specific type of network requests outstanding. Furthermore, this threshold number should be some number that is set in accordance with the current network settings and I/O workload. Determining sophisticated parameters for indicating that there is some significant outstanding network request that would benefit from a worldswitch is not immediately obvious.

Currently, my implementation requires pinning each guest to a physical CPU. I believe that this actually reduces the overhead in the system, but this should be verified by comparing the current implementation with worldswitching guests that are not pinned. Altering the current implementation to allow guests to float between CPUs as in the default configuration of Xen would not be difficult, but requiring guests to be pinned was the simplest way to implement the mechanism.
As discussed previously, the worldswitch table was developed as a general and flexible framework that could be used for a variety of purposes. However, this thesis used the table to allow all guests on the system equal access to the network driver. It would be useful to experiment with using the table only to experiment with setting different policies for different guests. For companies such as Amazon that sell virtual machine instances by the hour, it is valuable to be able to set different policies for different virtual machines and have those policies enforced.

Finally, a direction for future investigation is to combine the worldswitch mechanism with the mechanisms of Ram et al. [31] and Santos et al. [33] in an attempt to allow Xen guests to achieve highly efficient performance. Specifically, by combining the work in this thesis with their methods for moving packet demultiplexing out of the driver domain and creating an I/O translation table to track existing memory grants, the need for scheduling the driver domain at all could be eliminated. Because the receive interrupt for a guest’s dedicated receive queue would be delivered to the guest rather than to the driver domain, the guest itself would be scheduled in response. The guest could then use the worldswitch mechanism to retrieve the packet from the driver domain.
Bibliography


[16] Diwaker Gupta, Ludmilla Cherkasova, Rob Gardner, and Amin Vahdat. En-
forcing Performance Isolation Across Virtual Machines in Xen. Lecture Notes in

Twelfth ACM Symposium on Operating Systems Principles, pages 91–101. ACM,
1989.

[18] David B. Johnson and Willy Zwaenepoel. The Peregrine High-Performance RPC

[19] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High Per-
formance VMM-Bypass I/O in Virtual Machines. In Proceedings of the USENIX

Real Hardware, Virtual Hardware, and Other Virtual Machines. IBM Systems


