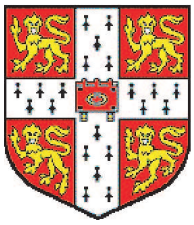
Virtual Devices for Virtual Machines

Andrew Kent Warfield

Clare Hall



A dissertation submitted to the University of Cambridge
for the degree of Doctor of Philosophy

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UK

Email: andrew.warfield@cl.cam.ac.uk

May 5, 2006

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.
This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.
No part of this dissertation has already been, or is currently being submitted for any such degree, diploma or other qualification.
This dissertation does not exceed sixty thousand words.

Virtual Devices for Virtual Machines
Summary

Andrew Kent Warfield
Clare Hall College

May 5, 2006

Computer systems research has recently seen a huge resurgence of interest in hardware virtualization, a software technique originally developed to manage mainframe computers in the 1960s. Using virtual machines (VMs), a commodity PC may be divided into isolated "slices", each perceiving that it is executing on separate physical hardware. This thesis considers the effective virtualization of I/O devices on commodity hardware and presents an approach that allows developers to add new functionality to a piece of hardware as a software extension, running in an isolated VM. The new virtual device is presented to the OS using the existing virtualized hardware interface, allowing extensions to be easily applied across a wide range of operating systems.

Isolating extensions in their own virtual machines is effectively a "sledgehammer" version of the system decomposition that was attempted by microkernels through the 1980s and 1990s. The VM-based approach has the benefit of demonstrably working with a broad range of existing systems, and allowing developers to build extensions in their OS and language of choice. It concurrently maintains the benefits of isolation: extension crashes are protected from disrupting the rest of the system, and extension software has a clean and simple interface to devices. This thesis develops this work by demonstrating the construction of a set of device extensions for various pieces of hardware. Additionally, this thesis demonstrates that device extensions may be aggregated within cluster environments to implement *device services*, allowing specific device types to be treated as a service throughout a cluster of virtual machines.

Several examples are presented to validate the flexibility of device extensions: A packet symmetry-based rate limiter demonstrates a single-host network extension that prevents VMs from issuing common forms of denial of service attacks. Parallax, a distributed storage system for VMs, demonstrates the implementation of a device service for the management of storage within a cluster. Finally, device extensions are combined with other virtualization projects to develop deployable system-wide extensions to virtual hardware.

Acknowledgements

I am indebted to my advisor, Steven Hand, who has acted as both friend and mentor throughout my doctoral work. Steve is a gifted teacher and frequently provided the right guidance at the right time. I hope to be able to match his standard in supervising my own graduate students in the future.

The packet symmetry work in Chapter 4 stems from collaboration with Christian Kreibich, Jon Crowcroft, Steve Hand, and Ian Pratt. James Bulpin developed two excellent implementations of the distributed block store used by Parallax in Chapter 5, and Christian Limpach provided the file system measurements presented in Figure 5.6. The work in Chapter 6 is a result of collaborations in combining the soft device framework with other research projects. The extensions for debugging are in collaboration with Alex Ho, while taint tracking is the result of ongoing collaboration with Alex, Michael Fetterman, Christopher Clark, and Steve Hand.

I have been very lucky to be a member of the Systems Research Group during a period of time where so much great work has been in progress, and am delighted to have been able to participate in such a broad range of projects during my time at Cambridge. In particular, James Bulpin, Julian Chesterfield, Christopher Clark, Jon Crowcroft, Michael Dales, Tim Deegan, Michael Fetterman, Keir Fraser, Alex Ho, Eva Kalyvianaki, Christian Kreibich, Christian Limpach, Anil Madhavapeddy, Rolf Neugebauer, Ian Pratt, Russ Ross, and Steven Smith have all provided insightful discussion and generally been really good fun to work with.

Thanks to Steve, Jon, Christian K., Julian, Rolf, Tim D., Alex, and Richard Mortier for providing feedback on drafts of this thesis.

Thanks finally to the staff of the Castle Pub, who have provided the impetus for more interesting research than they can possibly realize.

Table of Contents

List of Tables

List of Figures

Glossary

ABI	Application Binary Interface
CVD	Cluster Virtual Disk
IPI	Inter-processor Interrupt
OS	Operating System
VCPU	A Virtual CPU
VM	Virtual Machine
VMM	Virtual Machine Monitor

Backend	VM Containing a backend driver. (§3.2)
Backend Driver	A device driver that maps requests from frontend drivers in client VMs onto a physical device driver in the backend VM. (§3.2)
Communication Ring	A shared memory ring used to pass messages between VMs. (§3.2.3.4)
Device Tap	A driver that allows interposition and redirection of device requests. (§3.3)
Device Channel	A mechanism for high-performance inter-VM communication based on the combination of communication rings and event channels. (§3.2.3)
Device Service	An aggregation of a set of soft devices, allowing a given device class to be managed as a service across a set of physical hosts.
Domain	Logical container within which a VM executes.
Domain 0	Privileged domain, capable of administering other domains.
Driver VM	VM, with hardware access to a device, which hosts a physical device driver. (§3.2)
Event Channel	One-bit VM-to-VM notification mechanism; effectively a virtual interrupt line. (§3.2.3.1)

Event Notification	Notification received on an event channel. (§3.2.3.1)
Extension VM	VM hosting a device extension. (§3.3)
Foreign Page	A page of memory mapped from another VM.
Frontend Driver	A simple virtual device driver allowing a VM to access a class of devices (e.g. disks, network interfaces) exported by a backend driver. (§3.2)
Grant Mapping	A grant table operation allowing a page of memory belonging to one VM to be mapped into another VM as shared memory.
Grant Table	A per-VM table that lists pages of memory that the VM is sharing (mapping) or transfering to other VMs.
Grant Transfer	A grant table operation allowing a page of memory belonging to a VM to be relinquished, and given to a specific other VM.
Guest	A VM running above a hypervisor.
GuestOS	The OS used in a guest. In the case of Xen, guestOSes are often *paravirtualized*.
Hypercall	Analogous to a system call, a hypercall is a request for the hypervisor to perform a privileged operation.
Hypervisor	*syn.* Virtual Machine Monitor
Machine Memory	In the context of a VMM, machine memory refers to the real hardware memory on the system.
Page	Unit of memory described by virtual memory hardware on a processor. Notwithstanding processor extensions, a page of memory on the x86 architecture is 4096 bytes.
Paravirtualization	A technique of hardware virtualization in which the virtual hardware interface is different from the real physical hardware. Paravirtualization is used to improve performance, but requires that hosted operating systems be modified to reflect the modified hardware interface.
Physical Memory	*syn.* Pseudo-physical memory.
Pseudo-physical Memory	Virtualized machine memory, presented as physical memory to a VM.
Soft Device	A device, presented to a virtual machine, which has had its functionality augmented in software. (§3.3)
Split Driver	A client-server-style device driver model used to share access to I/O devices in a VMM. (§3.2)
Tap	*syn.* Device Tap.

Virtual Machine	An execution environment in which hosted software is presented with the illusion that it has ownership of a complete physical machine.
Virtual Machine Monitor	A piece of low-level systems software that multiplexes access to physical hardware across a collection of *virtual machines*.
Xen	A virtual machine monitor developed at the University of Cambridge.
XenBus	A device driver used to map device configuration information from XenStore onto an OS's device probing interfaces. (§3.2.4.1)
XenLinux	The *paravirtualized* version of Linux that runs on Xen.
XenStore	A persistant hierarchical store used to communicate configuration data in Xen. (§3.2.4.1)

Chapter 1

Introduction

1.1 Motivation

The development of system software to support access to I/O devices is a source of many challenges. For years, OS developers have been tasked with efficiently bridging the gap between the low-level interfaces of a constantly evolving set of device hardware, and the semantically richer, high-level interfaces required by application software. While the design space between these two interfaces is broad and allows considerable room for innovation, it has resulted in a wide variety of OSes for common, commodity hardware that each have a different and generally incompatible approach to interacting with devices.

Device driver code is hard to write, and has been described as the most error-prone subset of modern operating systems [CYC+01, SBL03]. As the driver-OS interface is not standard across systems, device vendors are unable to develop robust commercial drivers for every OS on which a driver will be used. The lack of driver availability and the difficulty in supporting devices as they emerge have been described as a stifling factor in the development of new OSes [FBB+97].

As the task of supporting devices on an OS is difficult, the act of extending them—installing software to augment the functionality of a given device—is very hard indeed. Despite the fact that many interesting research projects have shown the power of innovation at the device interface to build extensions such as secure [GNA+97], distributed [LT96, SFV+04], or versioned [WCG04] storage and packet filtered [EK96, PF01] or intrusion-detecting [WCSG04] network interfaces, these efforts have had only min-

imal impact on real systems. Moreover, development of these extensions is complicated by the eccentricities of individual OS device interfaces, and the resulting code is invariably hard to maintain or port to other systems.

As commodity systems have become increasingly powerful, and organizations have become concerned with the degree of *utilization* of individual physical servers, there has been a renewed interest in hardware virtualization, a technique originally developed for mainframes in the late 1960s which allows a physical host to be partitioned into a number of *virtual machines* (VMs). The availability and growing deployment of hardware virtualization on commodity systems has interesting implications for problems relating to device management in systems software; virtualization results in both challenges and opportunities in managing devices.

On one hand, hardware virtualization presents a challenge in that it is unsafe to grant concurrent access to a device's physical interface to more than one operating system. A system providing virtualization is thus responsible for safely multiplexing low-level device access among running virtual machines. Not only must VMs be able to share access to physical devices such as disk and network, but they must be prevented from interfering with one another either maliciously or accidentally.

Conversely, virtualization presents developers with the ability to interact with devices *below* the OS's hardware interface. As virtualization software runs underneath OS code, interactions with virtualized device hardware must pass through it. This unique position allows the introduction of device extensions that are inherently both isolated from and portable across the range of OSes that run in the virtualized environment.

1.2 Contribution

It is the thesis of this work that a well-designed approach to device virtualization addresses the major problems of portability and extensibility in the management of devices on commodity systems. Using the Xen virtual machine monitor, a widely-available and robust VMM that has been developed at Cambridge over the past four years, this work demonstrates a set of approaches to the virtualization and extension of I/O devices for commodity

hardware. The techniques described are validated through the development of a set of practical, useful device extensions targeted primarily at large computer installations, such as data centres, where virtualization is used.

The initial contribution of this work is the design and implementation of an architecture for the development of device extensions. The combination of a physical device and extension software that modifies the behaviour of that piece of hardware is described as a *soft device*. I have constructed a set of software tools that allow the construction of soft devices for the disk and network device interfaces that exist in Xen today. This approach demonstrates that extensions may be written and executed in user-space of an isolated VM, allowing developers complete freedom to innovate while maintaining reasonable performance.

The second contribution of this thesis is the aggregation of soft device-based extensions to form *device services*. Device services allow device extensions to be composed into cluster-wide facilities which serve large numbers of virtual machines.

To demonstrate the range, scope and flexibility of these techniques, I have explored the construction of both soft devices and device services for a variety of applications. An additional contribution of this thesis is the exploration of a set of such examples, specifically storage, traffic management, and whole-system extensions, that are targeted to address relevant problems in clustered VM environments.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2 is a discussion of the relevant background and aims to familiarize the reader with the current state of hardware virtualization in general and Xen in particular.

Chapter 3 describes the complete set of mechanisms for device virtualization and extension advocated by this thesis. It begins with a detailed presentation of the *split driver* model for providing virtual devices in Xen. Next it presents the soft device architecture for constructing isolated device exten-

sions, and demonstrates implementations of extension support for both disk and network devices. The chapter concludes with a description of the device service model for aggregating device extensions in cluster environments. The remaining chapters of the thesis validate this architecture by demonstrating examples of soft device-based extensions.

In Chapter 4, I discuss extensions for the support of network interfaces. After surveying the challenges faced in virtualizing network devices, the chapter presents an example device extension, a packet symmetry-based rate limiter. This extension monitors outbound and inbound packet counts and prevents VMs from being used maliciously to mount denial of service attacks.

Chapter 5 presents Parallax, an example device service to address the storage requirements of virtualization-based clusters.

Chapter 6 presents a final example of the application of device services. This chapter considers the combination of the techniques developed in this thesis with more sweeping changes to the virtualization system in order to realize full-system architectural change. The chapter presents two examples of such change: extensions to support whole-system debugging, and taint-based memory protection.

Chapter 7 places this thesis in the context of relevant related work and Chapter 8 concludes and discusses directions for future investigation.

1.4 Published Results

Some aspects of this work have been described previously. In reverse chronological order, the list of related publications is as follows:

1. C. Kreibich, A. Warfield, J. Crowcroft, S. Hand and I. Pratt. Using Packet Symmetry to Curtail Malicious Traffic. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, College Park, MD, 2005.

   Describes initial results regarding the packet symmetry scheme presented in Chapter 4.

2. A. Warfield, R. Ross, K. Fraser, C. Limpach and S. Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, 2005.

   Presents the Parallax prototype, which is extended by the work described in Chapter 5.

3. S. Hand, A. Warfield, K. Fraser, E. Kotsovinos and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, 2005.

   Argues that virtual machine monitors provide a practical means of achieving the isolation goals sought by microkernel research. The theme of this paper represents the nucleus of the argument presented in this thesis.

4. C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, 2005.

   Presents the design and implementation of live-migration support for Xen-based virtual machines. The ability to migrate running virtual machines punctuates the separation from real hardware that exists in these environments, and helps form the basis of the argument for device services presented in Chapter 3, and illustrated by Parallax in Chapter 5.

5. A. Warfield, S. Hand, K. Fraser and T. Deegan. Facilitating the Development of Soft Devices. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, 2005.

   Discusses the initial implementation of the block tap, an instance of a device tap as presented in Chapter 3.

6. K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS-*

1), Boston, MA, 2004.

Motivates the use of virtual machine monitors to enhance the reliability of legacy device drivers for commodity systems, and explains the driver architecture used to share I/O devices across virtual machines in Xen. This paper presents an initial discussion of split drivers, which are detailed in Chapter 3.

7. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, 2003.

   Provides a detailed technical description of the initial public release of the Xen virtual machine monitor.

8. A. Warfield, S. Hand, T. Harris and I. Pratt. Isolation of Shared Network Resources in Xenoservers. *PlanetLab Design Note PDN-02-006*, 2002.

   Describes early approaches taken to the virtualization of network devices. This paper predates the driver architecture that is used in this paper.

Chapter 2

Background

This chapter presents background material relevant to this thesis. The material here is not a comprehensive presentation of related work, which is discussed at the end of this thesis in Chapter 7. Instead, it is intended to familiarize the reader with the relevant aspects of hardware virtualization and some specific details of the Xen virtual machine monitor. While hardware virtualization is a long established technique in systems research, it has only returned as a common approach on commodity systems over the past few years. This chapter attempts to set the stage for the remainder of the thesis, first by explaining why virtualization has recently become important again and second by detailing specific aspects of virtualization as realized by Xen.

After describing the design and implementation of Xen, the chapter ends with a presentation of live VM migration. Live migration allows a running virtual machine to be relocated to a new physical host and has been implemented on Xen. Migration illustrates the ability of hardware virtualization to provide useful new OS-agnostic features, and serves as a motivating example for the device-specific extensions that are presented throughout the following chapters.

2.1 The Virtualization Renaissance

A virtual machine monitor (VMM) is a narrow layer of software that provideѕs a set of isolated execution environments that closely match the underlying physical computer. Each of these environments is called a virtual machine (VM), and may contain an operating system and associated set

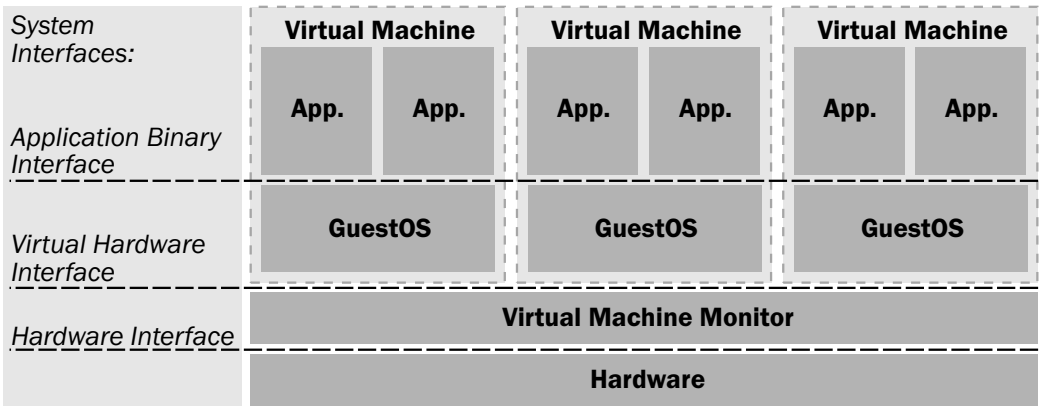


Figure 2.1: Overview of a VMM-based System

of applications. A major aspect of virtualization is the focus on maintaining high performance relative to the underlying hardware. As such, in his 1972 thesis, Goldberg distinguishes virtualization from more heavy-weight techniques such as emulation and simulation by observing that a VM is "a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode" [Gol72].

The technique of virtualizing hardware dates back to the 1960's, when it was used to divide resources on mainframe computers into coarse-grained partitions. Over the past few years, VMMs have returned as an attractive technique on commodity systems for many of the same reasons that they were initially deployed on mainframes. The core property achieved by a VMM is that of *isolation*; VMMs allow a powerful physical machine to be partitioned into a set of virtual machines, each of which may be separately scheduled, configured, administered, and otherwise managed. The importance of coarse-grained isolation in a VMM environment cannot be overstated, and the point will be revisited throughout this chapter as it is fundamental to the argument made by this thesis.

2.1.1 Role of a Virtual Machine Monitor

Stated briefly, a VMM provides a rigid partitioning of the low-level hardware resources that are typically managed by an operating system. By dividing the system at such a low level, OSes may continue to be deployed with

little or no change; the VMM simply allows a multiplicity of OSes to share a common physical host. Additionally, as the resources divided by a VMM are simple, its complexity is considerably lower than that of an OS. Unlike OSes, VMMs maintain a minimum of state, and are solely responsible for the enforcement of resource partitioning within a system.

The small size of VMMs, and their use of simple hardware interfaces makes them attractive for a variety of roles in the management of systems. These roles are largely the same today as they were in their deployments on mainframes. Goldberg's 1974 article in IEEE Computer, "Survey of Virtual Machine Research" [Gol74] describes a variety of applications of VMMs, including the hosting of legacy applications that are "locked" to a specific operating system, facilitation of OS development and testing, and the testing and debugging of networks and distributed systems by virtualizing them on a single physical host.

2.1.2 VMMs in Modern Systems

VMMs have become attractive again in modern systems for reasons similar to those which motivated their application on mainframes: Commodity harware has become sufficiently powerful to host multiple virtual machines on a single physical host, while the benefit of virtualization to administrators justifies deployment in many environments.

2.1.2.1 *Cost*

Why has there been such a sudden resurgence of interest into the application of virtualization on commodity systems? As with many techniques in software systems, the decision to use virtualization is largely based on considering the trade-offs between performance overhead and functional benefit. In addition to the overhead of virtualization itself, the introduction of a VMM requires that a system be sufficiently powerful as to be divided into a set of virtual machines, each of which will run as if it were a single physical host. Each "slice" of the system must have sufficient resources to reasonably run an operating system and its applications. In the case of Xen, which is discussed in more detail in Section 2.2, the physical machine is intended to be divided into up to 100 VMs, and the overhead of virtualization is demonstrably very low. Using VMMs such as Xen, commodity servers easily have

sufficient resources to be divided into a small number of virtual machines, affording this *cost* of virtualization. This fact was initially demonstrated in our paper on Xen [BDF+03], in which commodity server hardware is partitioned and VMs run industry-standard benchmarks in parallel.

2.1.2.2 *Benefit*

The evolution of modern hardware has reached a point at which the deployment of virtualization is practical. As such, it is worth considering the *benefits* provided by virtualization that are motivating its adoption. As mentioned above, the key benefit provided by a VMM is *isolation*. Independent of virtualization, modern systems have adopted a model in which individual server applications are often each installed onto a separate physical host. This one-to-one mapping between software servers and physical servers is largely a result of the tight coupling between server applications and the OSes that they run on: isolating servers onto separate physical machines avoids undesirable interactions between configuration and administration of the host. The unfortunate consequence of this trend is that server rooms are increasingly full of mostly idle physical machines: a study by IBM published in 2003 reported that the average daytime utilization of Windows servers across an organization was less than 5%, and 15-20% for UNIX servers [Hea03].

The desire to address wasted physical resources—resources such as space, cooling, and power consumption—has been largely responsible for the revitalization of virtualization research and development. Large organizations are increasingly moving towards a utilization-based approach to management, and are using virtualization to consolidate servers onto smaller numbers of physical hosts.

While this trend is likely the largest overall motivation for the renewed interest in virtualization, there are a host of other benefits offered by VMMs on modern systems. Applications such as testing and development, and support for legacy applications and heterogeneous OSes, which were major applications of VMMs in the sixties and seventies [Gol74] still apply. Moreover, researchers are using VMMs for a variety of new applications such as intrusion detection and forensics [DKC+02, JX04, VMC+05], configuration management and debugging [WCG04], and software debugging [KDC05, HH05], to name only a few.

2.1.2.3 *Challenge*

While the deployment of VMMs is justified in terms of the trade-off between overhead and benefit, the development of functional VMMs for commodity systems has been very challenging. Unlike the mainframes on which virtualization was initially deployed, the x86 architecture was not designed with virtualization in mind. The processor is not inherently virtualizable, and requires work-arounds in software to make virtualization efficient. Section 2.2 details the design of Xen, and the challenges presented in virtualizing the x86 architecture.

2.1.3 High-level Principles

Prior to the technical overview of Xen, it is worth emphasizing two high-level aspects of virtualization design that have emerged as significant insights through work with Xen. The two points presented here have had a great deal of influence on the design and development of the work described in this thesis, and it is useful to consider them further.

2.1.3.1 *The Value of Coarse-Grained Isolation*

As mentioned above, the isolation provided by VMMs is provided at a low-level and is both very strong and very coarse-grained. VMs are assigned a partition of system resources including disjoint regions of physical memory and storage, and have strongly enforced CPU allocation. It is important to observe that this isolation is not limited to the performance-centric division of resources: As alluded to throughout this chapter, VM-based isolation is very near to that of running on a separate physical host. As such, isolation also applies to properties such as configuration, administration, and security.

As the interface between the VM and the VMM is very narrow, the VMM itself maintains very little per-VM state. This property has enabled functionality that has been incredibly difficult to achieve in modern operating systems, where the OS stores considerable per-process state. Live VM migration, discussed later in this section, is an example of such a benefit: While process migration has typically been very difficult to achieve and is not provided by any modern OS, VM migration is relatively simple to implement

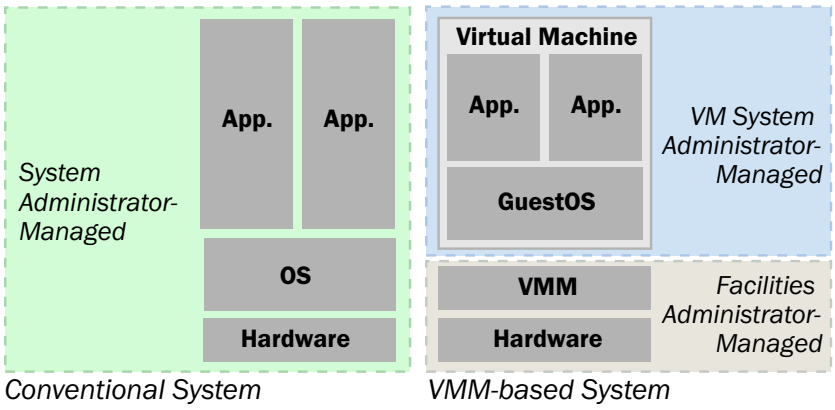


Figure 2.2: Division of the Administrative Role

and is supported by several existing VMMs.

The isolation provided by VMMs has been likened to the isolation advocated between system components by microkernels, with some VMM designers having gone so far as to describe their own systems as microkernels [WCSG04] while some microkernel researchers describe their systems as VMMs [LUSG04]. A discussion of the success of the VMM approach, relative to that of microkernels is presented in [HWF+05]: In brief, we propose that although the isolation and architectural elegance provided by microkernel systems is a noble goal, it fails to recognize the demands of existing systems running existing applications. By treating the OS as a functional unit and maintaining application compatibility within OSes, VMMs have been remarkably successful in impacting real systems. The design of VMMs and VMM-based extensions is founded on the treatment of VMs largely as black boxes, and focusing on the management of low-level resources and entire operating systems.

2.1.3.2 *The Division of the Administrative Role*

In addition to the isolation between individual VMs provided by a VMM, it is worth considering the impact of the horizontal division that is introduced between a VM and the hardware that it runs on. A major impact of the introduction of a VMM is that it bisects the management of a physical host, and the application of policy decisions within the system as a whole.

In a non-virtualized system, there is effectively a single administrator. All software-based management requires administrative access to the OS. However, physical administration also requires this access in order to safely shut down and restart machines that are being serviced. The system administrator is concurrently responsible for enabling functionality on a host, by installing and configuring new software, and for policing the behavior of that host, for instance by preventing its use for malicious intent.

As illustrated in Figure 2.2, the introduction of a VMM divides this administrative role in two. As isolated units, VMs are administered by individuals responsible for the software that is installed on them. This administration involves traditional OS management activities such as installing software, configuring application and OS configuration, and managing users. All of these management tasks are applied to a virtual machine, and the administrator is unconcerned with the physical hardware that underpins that VM save for the fact that it works as expected. Policy decisions made at the VM level concern high-level primitives, such as files and users, that exist within the context of that virtual host.

The physical hardware then is managed by a second, lower-level *facilities administrator*. This role is concerned with the upkeep of physical hardware, and the *enforcement* of low-level policy within the system. The isolation provided by the VMM reflects the sort of low-level policy enforcement that the facilities administrator is concerned with: VMs should receive their expected share of resources, they should be prevented from interfering with other hosts or from otherwise impairing the overall physical environment. The isolation provided by the VMM is particularly beneficial in this sense in that the consequent ability to migrate running VMs between physical machines enables low-level administration without requiring access to individual virtual machines.

In short, modern systems have demonstrated both the capacity and the need for the deployment of hardware virtualization. The adoption of virtualization is a powerful tool that allows for strong isolation between OSes, and the division of the administrative role. In some senses, the move toward virtualization is a return to the centralized physical resource management that was provided by mainframes, while continuing to take advantage of commodity hardware, and allowing distributed software administration.

The remainder of this chapter presents an overview of the Xen virtual machine monitor, which has been developed at the University of Cambridge over the past four years and on which the remainder of this thesis is based.

2.2 The Xen Virtual Machine Monitor

While there is a demonstrable case for the application of virtualization techniques on commodity hardware, the development of effective virtual machine monitors for such systems is a non-trivial task. In a 1974 paper, Popek and Goldberg describe a set of formal requirements for hardware systems to support virtualization [PG74]. The key observation of their work is that in order to directly support virtualization, all instructions which access privileged system state must result in a trap when executed outside of supervisor mode. This requirement provides the VMM the opportunity to safely validate and issue privileged operations on the behalf of individual virtual machines, as the operating systems within these VMs no longer execute directly in supervisor mode.

The x86 architecture does not meet the criteria set out by Popek and Goldberg. Specifically, the x86 instruction set contains privileged instructions which, when executed in user mode, do not generate a trap and so do not allow the VMM to intervene. For example, the `PUSHF` and `POPF` instructions allow the contents of the processor state flags register (`EFLAGS`) to be loaded to and from the top of the stack. The `EFLAGS` register is used by applications to identify conditions such as arithmetic overflow. However, it is also used in supervisor mode to modify aspects of processor state, for instance to enable and disable interrupts. As such, an unmodified OS binary that is moved out of supervisor mode (protection ring 0) with the introduction of a VMM will silently fail to update `EFLAGS`, and will not behave correctly. Robin and Irvine [RI00] present a detailed survey of the Intel Pentium instruction set, and identify seventeen such non-virtualizable instructions.

An additional consideration in the design of a VMM, described by Goldberg [Gol72], relates to the placement of the VMM itself. Goldberg attempts to dichotomize VMM architecture by differentiating between a VMM that

Goldberg identifies two architectural approaches to the design of VMMs. A Type I VMM executes directly on raw hardware, while a Type II VMM runs as an application within a Host OS.

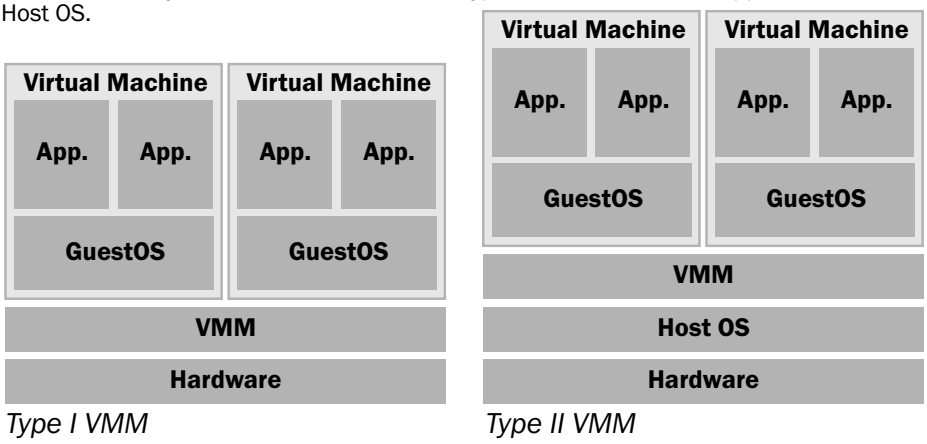


Figure 2.3: Type I and II VMMs

runs directly on hardware, dubbed a Type I VMM, and a Type II VMM, which run as an application within an existing "Host OS". This distinction is shown in Figure 2.3: By moving the VMM itself into application space, it may defer the complexities of scheduling, memory management and device access to the Host OS. However, such Type II VMMs are limited in terms of their ability to provide isolation and resource control to the mechanisms offered by the Host OS to its applications.

This taxonomy was extended by Popek and Goldberg to address the virtualization of certain non-virtualizable architectures [PG74]. The authors introduce the notion of a *hybrid VMM* (HVM), which emulates all supervisor-mode instructions issued by the virtual machine. This model capitalizes on the ability to trap state transitions between user and supervisor mode to activate an instruction emulator, and achieves correctness by sacrificing the performance of native execution for VM kernel code. It is worth noting that many commercial VMM products, such as VMware Workstation, are essentially Type II HVMs. However, the taxonomy is hardly a rigid one, in that these products generally both install kernel extensions into the Host OS and use some combination of binary scanning and code rewriting to reduce the need for instruction emulation in the VM kernel.

2.2.1 Xen

The Xen virtual machine monitor is a VMM, originally for the Intel x86 architecture, developed at the University of Cambridge. Xen was motivated by the desire to provide a high-performance VMM with strong resource isolation properties on commodity hardware. As such, the design was for a Type 1 VMM, where hardware could be more rigidly controlled.

A key distinction between Xen, and existing commercial VMMs is the introduction of *paravirtualization*. A paravirtualizing VMM addresses the same problem as the hybrid VMM mentioned above, but the design optimizes for performance rather than interface preservation: Instead of emulating supervisor-mode instructions, a paravirtualizing VMM mandates that those instructions are not directly used by the VM. VMMs such as Xen and Denali [WSG02] present a modified hardware interface in which non-virtualizable instructions must be specifically issued to the VMM as *hypercalls* (hypervisor calls), allowing them to trap to the VMM and be validated and issued on behalf of the VM.

Xen is different from Denali by merit of the fact that it preserves the *application binary interface* (ABI). While OSes themselves must be specifically ported to run on a paravirtualizing OS, the user-mode interface is preserved such that application binaries need not be recompiled in the case of Xen. As with much of Xen's design, this was a practical decision: by maintaining ABI compatibility, existing OS distributions may be installed directly into VMs, needing only a Xen-supporting (e.g. XenLinux) kernel. Denali's application interface is markedly different from that offered by hardware: The VMM allows only single-address-space, single-threaded applications which have been linked directly against the Ilwaco guest OS. Additionally, Denali does not preserve aspects of the hardware interface such as segmentation, which are used by many applications.

Over the past four years of development, the Xen project has been very successful. The VMM has been extended to support other architectures including both Intel's Itanium (IA64) and the 64-bit x86 (x86-64) processors. Additionally, support has been added for emerging hardware virtualization extensions, allowing the VMM to support unmodified OSes without requiring paravirtualization.

Memory Management	
Segmentation	Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.
Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.
CPU	
Protection	Guest OS must run at a lower privilege level than Xen.
Exceptions	Guest OS must register a descriptor table for exception handlers with Xen. The handlers remain the same.
System Calls	Guest OS may install a 'fast' handler for system calls, allowing direct calls from an application into its OS instance and avoiding indirecting through Xen on every call.
Interrupts	Hardware interrupts are replaced with a lightweight event system.
Time	Each guest OS has a timer interface and is aware of both 'real' and 'virtual' time.
Device I/O	
Network, Disk, etc.	Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.

Table 2.1: The Paravirtualized x86 Interface.

2.2.2 Paravirtualized Hardware Interface

Xen's paravirtualized hardware interface, summarized in Table 2.1, is based on four design goals: First, it addresses the non-virtualizable aspects of the x86 architecture. Second, it aims to minimize virtualization overhead, opting to modify the hardware interface where there is a strong performance benefit to doing so. Third, it maintains the application binary interface so that existing application binaries may execute unchanged. Finally, it attempts to keep the degree of change required to a Guest OS to a minimum, in order to facilitate the porting of new systems to Xen.

The design of Xen has evolved considerably over the past four years. This section presents an overview of Xen's design with regard to the aspects of the system described in Table 2.1. The next chapter presents a detailed discussion of the virtualization of device access in the VMM.

2.2.2.1 *CPU*

The x86 architecture offers a ring-based protection model. While the processor has four levels of hardware protection, conventional OSes use only two: in general the OS runs in the most privileged ring (ring 0), and applications run in the least privileged (ring 3). With the introduction of Xen, the hypervisor executes in ring 0, and the guest OS is moved to ring 1. This approach maintains protection across layers in the system, while allowing the hypervisor to retain complete control. As discussed above, non-virtualizable instructions in the guest OS no longer function as intended when removed from ring 0 and must be replaced with hypercalls to Xen.

Each virtual machine is configured with a set of virtual CPUs (VCPUs), which represent the unit of scheduling that is managed by the VMM. There need be no correlation between the number of physical CPUs in a system and the number of VCPUs given to a VM. Virtualizing the processor in this manner allows a SMP system to be divided into a set of isolated uniprocessor VMs, and conversely allows the testing of SMP OS code on a uniprocessor system. The hypervisor provides a variety of schedulers, and allows individual VCPUs to be "pinned" to specific physical CPUs.

All processor exceptions are handled by Xen. Guest OSes register a table of exception handlers which Xen will validate and install on their behalf. System calls, which are relatively common, and hence could represent a high performance overhead, are handled by allowing the guest to register a fast trap handler which results in the CPU calling directly into the guest OS.

Page faults represent an example of a situation where paravirtualization is not absolutely required, but serves to greatly improve performance. On x86 page faults, the faulting linear address is reported in a register (`CR2`) which may only be read in ring 0. Xen preserves this value so that it may be accessed in the OS fault handler in ring 1. A non-paravirtualizing solution would be to pass control to the OS fault handler, which would then immediately trap on the attempt to read `CR2`. Xen could then emulate the access,

returning the stored value from the original fault. The performance impact of this approach would be very high, as each page fault would require two entries into the VMM. Instead, the page fault handler in Xen writes the faulting address into a shared-memory location associated with the faulting virtual CPU. Control is then returned to the guest fault handler, which may read the value from shared memory instead of `CR2`.

Interrupts are completely virtualized and a guest OS never runs with hardware interrupts disabled. Each virtual CPU has a set of 1024 virtual interrupts, or *event channels*. These may be mapped to physical interrupts or connected to virtual interrupts on other VCPUs, even those on other VMs. As such, event channels allow virtual interrupt and interprocessor interrupt (IPI) facilities, and also provide a capability to generate interrupt-like notifications *between* virtual machines. Guest OSes receive event notifications through an upcall from the hypervisor while they are executing. In addition, the hypervisor provides a virtual timer event every 10ms while a VM is scheduled. The event channel interface is described in more detail in the next chapter.

2.2.2.2 *Memory Management*

The VMM resides at the top of virtual memory, and is protected using x86 segmentation. This approach allows the VMM to have a constant location in memory regardless of execution context, and avoids unnecessary TLB flushes on protection domain crossings. However, it places a slight limitation on the use of segmentation within VMs, as segments must not overlap the top of linear address space, where Xen is located.

Xen adopts the terminology used in Disco [BDGR97b] to describe memory as it is accessed from various contexts within the system. *Machine* pages refer to the hardware-addressable pages of memory within the host; they represent unvirtualized, untranslated memory within the system. *Physical* pages (also referred to as pseudo-physical pages) are the set of machine pages allocated to a VM. Physical memory is a virtualization of machine memory, and physical addresses are a contiguous range from zero to the amount of memory allocated to the VM. Each physical page refers to a machine page, but the underlying machine memory is not necessarily contiguous. Finally, *virtual* pages are references to machine memory through page table-based mappings, using the hardware MMU. Figure 2.4 demonstrates the

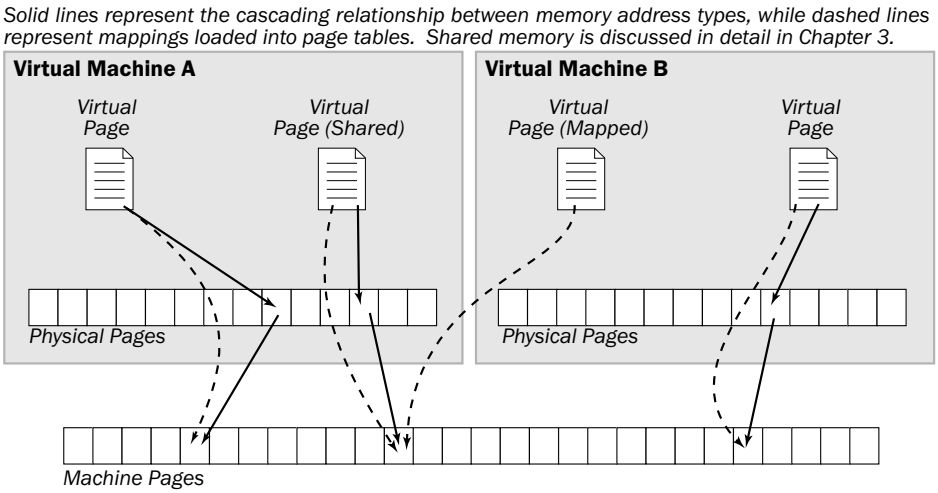


Figure 2.4: Memory in Xen

interactions between these memory contexts.

Machine pages, which are managed completely by the VMM, have type, ownership, and reference counts. A page of memory in the system may only be owned by a single VM, and has a single purpose. Typing ensures that the VMM is aware of all pages that are used as page tables, and can enforce safe access to memory management hardware.

Guest OSes maintain a table mapping their contiguous physical address space onto the associated underlying machine pages. By providing visibility to both physical and machine pages guests may efficiently manage their own memory. Typically, the memory allocator in a guest OS will manage ranges of physical memory, while page tables will be programmed directly by the guest with machine addresses, allowing native performance on virtual memory access.

Several techniques exist for the management of virtual memory under a VMM. The technique which is currently used by Xen is that of "writable page tables". In this approach, VMs have direct access to their own page tables. However, all page table pages are mapped read-only. On an attempt to update page tables, the resulting page fault will be handled by Xen and the page to be modified is "unhooked" from the parent table prior to granting write access. The guest may then make arbitrary updates to the page table page, but these updates do not apply until the guest attempts to ac-

cess virtual memory described by the unhooked page. At that point, Xen will handle the resulting trap and validate any changes prior to reinserting the page into the page tables and again marking it as read-only. The only exception to this technique is that of page table bases, which are described in a protected register (`CR3`) that is loaded using a hypercall. Writable page tables provide high performance, as collections of updates to a page table pages are implicitly batched, while imposing minimal change on the memory management code in the guest OS.

A second technique used for managing virtual memory in Xen, which is available as an alternative to writable page tables, is that of shadow paging. In this technique, the page table pages visible to a guest are never loaded into the MMU. Instead, the VMM maintains a shadow of these pages in hypervisor memory. Shadow paging has been used as an approach to memory management in VMMs for some time [Wal02, Gum83, HR91]. Although it imposes a higher overhead than writable page tables, the technique has many benefits. For instance, the set of machine pages underlying a guest may be modified while it is running, as the shadow page tables indirect the location of the machine page backing each virtual adress. A limited form of shadow paging is used to track page dirtying in live VM migration, described below, while a more complete version is used in the taint-tracking work described in Chapter 6.

2.2.2.3 *Device I/O*

Access to devices must be carefully managed in a VMM environment. OSes expect direct access to device hardware, and it is unsafe to allow multiple VMs to share concurrent hardware access to the same device. Xen protects hardware by allowing only a single VM to directly interact with a given device. The problem of multiplexing device access is addressed using *split drivers*: a single VM manages each physical device and multiplexes access to that device between other VMs.

The management of devices for virtual machines is the topic of this thesis, and the approach taken in Xen is discussed in great detail in Chapter 3. Device virtualization is responsible for providing the resource partitioning for resources other than processors and memory.

This section has summarized the set of approaches taken by Xen to paravirtualize the x86 architecture. This paravirtualization has resulted in the development of a stable, high-performance virtual machine monitor that is now used in many production environments around the world. The next section provides detail on a final aspect of Xen's impact on commodity systems, that of live VM migration. Live migration is worth discussing in some detail for two reasons. First, the ability to migrate a running VM between physical hosts has a major impact on how systems may be managed, and is an illustration of the attraction of VMMs to many organizations. Second, this thesis argues that hardware virtualization introduces an effective architecture with which to extend device functionality. Live migration provides an example that sets the stage for this argument by demonstrating the ability to implement a useful new feature *below* the operating system.

2.2.3 Live VM Migration

As mentioned above, virtual machine monitors maintain a narrow interface between the VM and the hypervisor and a very limited amount of per-VM run-time state when compared, for example, with managing processes within a conventional OS. As a consequence, it is possible to perform operations concerning the entire state of a VM—for instance suspending and resuming its state to and from disk—because there are very limited state-dependencies across the VM-VMM boundary.

Xen exploits this property, allowing a VM to be *migrated* to a new physical host while it runs. Live migration is a powerful management tool, in that it allows a facilities administrator to both balance load within a cluster, and to free up a physical host for hardware maintenance. In both of these cases, the administrator leaves the applications running within the VM undisturbed.

Live migration attempts to move most of the VM state to the new host while the VM continues to run. The only effect visible to the migrating VM is a brief pause in execution, as if it was descheduled, and the possible loss of a small number of in-flight packets. Migration times for server workloads result in service down times as low as 60ms.

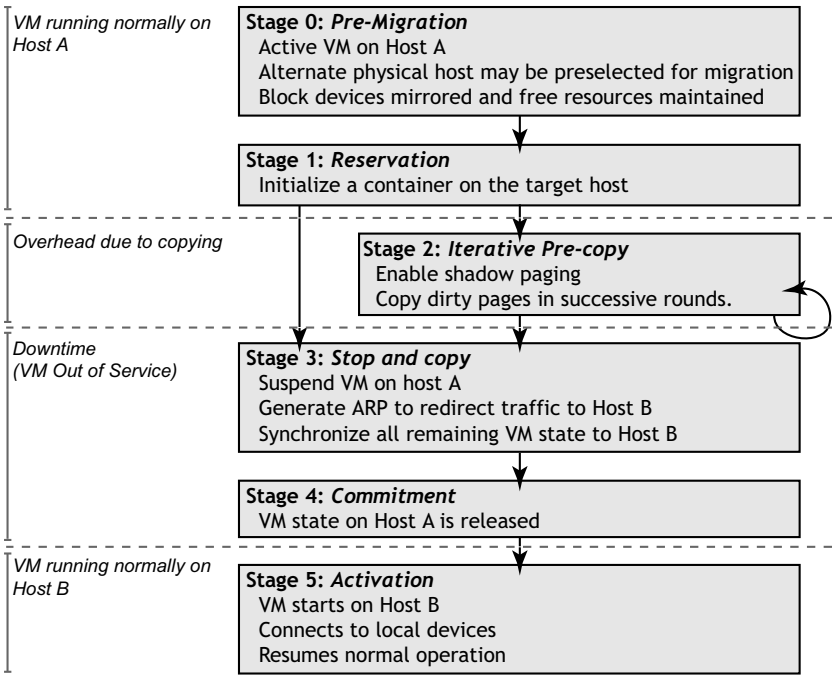


Figure 2.5: Migration Timeline

2.2.3.1 *Stages of Live VM Migration*

Figure 2.5 summarizes the migration of a VM between two physical hosts. This process is effectively a transaction that can be aborted at all but the final stage. Each stage in the figure proceeds as follows:

Stage 0: Pre-Migration. Migration begins with an active VM on physical host A. To speed any future migration, a target host may be preselected where the resources required to receive migration will be guaranteed.

Stage 1: Reservation. A request is issued to migrate an OS from host A to host B. The system confirms that the necessary resources are available on B and reserves a VM container of that size. Failure to secure resources here means that the VM simply continues to run on A unaffected.

Stage 2: Iterative Pre-Copy. All of the pages of the VM's memory are copied from A to B. A limited form of shadow page tables is used to track writes to copied pages, and the copy process iterates over the VM's

memory recopying any dirty pages. The rate of copying is compared against the rate of dirtying as an indication of progress, and iterative copy stops when progress diminishes.

Stage 3: Stop-and-Copy. The VM instance at A is suspended. Its network traffic is redirected to B by generating an unsolicited ARP advertisement that the interface now resides at the new host. CPU state and any remaining inconsistent memory pages are then transferred. At the end of this stage there is a consistent suspended copy of the VM at both A and B. The copy at A is still considered to be primary and may be resumed in case of failure.

Stage 4: Commitment. Host B indicates to A that it has successfully received a consistent OS image. Host A acknowledges this message as commitment of the migration transaction: host A may now discard the original VM.

Stage 5: Activation. The migrated VM on B is now activated. The VM restores device connections and continues to run as normal.

2.2.3.2 *Migration Performance*

Figure 2.6 demonstrates the effectiveness of live VM migration. In the experiment an Apache 1.3 web server is migrated between a pair of Dell PE-2650 servers. Each host has dual Xeon 2GHz CPUs and 2GB of memory, and the two are connected over a switched gigabit Ethernet using Broadcom TG3 interfaces. As the VM is migrating between physical hosts, network attached storage is used to achieve location transparent access to the system image, which is accessed using an iSCSI mount exported from a NetApp F840 storage server.

The web server VM is configured with 800 megabytes of memory, and is loaded with 100 clients, each continuously requesting a 512KB file. As shown in the figure, the server achieves an initial throughput of 870Mbit/sec. Migration begins, and is configured to use a maximum initial bandwidth of 100Mbit/sec. The first copy iteration can be seen to reduce the achieved bandwidth of web traffic during the approximately 60 seconds that are required to transfer the initial memory image. The copy process then iterates several times over memory during the following ten seconds before stopping VM execution and copying the final state. As shown, the copy throughput

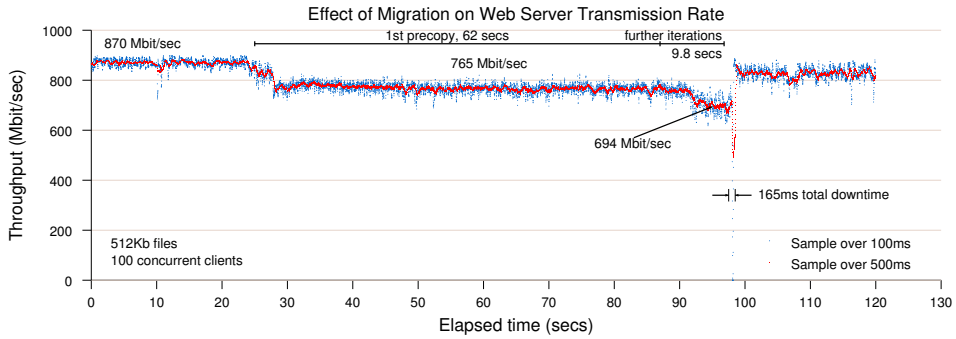


Figure 2.6: Results of Migrating a Running Web Server VM

increases during the iterative copy process, as the migration algorithm attempts to use extra bandwidth to match the page dirtying rate. The total downtime experienced by the server is 165ms, after which it returns to full performance on the new physical host.

A more detailed exposition of live VM migration is presented in [CFH+05], which discusses several other server examples and provides additional detail on the iterative copy algorithm. The intention of describing VM migration in this context has been to illustrate an example of the qualitative impact that VMMs provide in managing computer systems.

2.3 Summary

Hardware virtualization, initially developed in the 1960s for the management of mainframe computers has received renewed attention as a means of managing commodity computer systems. VMMs allow the resources of a physical host to be divided between a set of virtual machines, each of whom have the appearance of running on a single host. The key property achieved by virtualization is *isolation*: servers may be consolidated onto common physical hardware, and remain isolated from each other in all respects save the failure of physical hardware. This isolation is not limited to performance and resource control, as servers are also isolated in terms of configuration, security, and administrative control.

The administrative isolation achieved by VMMs is particularly relevant in modern systems, where organizations are attempting to centralize computing hardware and increase its utilization. The introduction of VMMs divides the administrative role in two parts allowing individual VMs to be administered directly by the responsible system administrators, while hardware may be administered by facilities administrators without requiring access to the contents of individual VMs.

Xen is a virtual machine monitor for commodity hardware that achieves the goals of hardware virtualization. Using *paravirtualization*—changing the exported hardware interface and requiring that guest OSes be explicitly ported—Xen overcomes the inherent problems of non-virtualizability presented by the x86 platform and also achieves drastic improvements in VM performance.

The availability of a robust VMM for commodity hardware is demonstrating the opportunities to achieve qualitative change on software systems by working *below* the operating system. An example of this type of change is live VM migration, which allows a running VM to be relocated to a new physical host with only milliseconds of downtime. Live migration exploits the coarse granularity of the VM, and the limited shared state that exist between it and the VMM. Moreover, the approach is independent of the contents of the VM being migrated and so applies to any operating system that runs on Xen.

Another example of the benefits of using virtualization to work at a low level within the system is that of device management. The next chapter presents the challenges and opportunities of designing and managing virtual device interfaces and presents a set of approaches for accessing and extending devices in VMM-based environments. The remaining chapters of the thesis validate this approach by demonstrating examples of virtual devices.

Chapter 3

The Soft Device Architecture

This chapter motivates and explains the core mechanisms proposed by this thesis, phrased as solutions to three successive architectural problems relating to I/O devices encountered in the design and deployment of VMMs. These problems are as follows:

1. **Device support in a VMM.** How should device access within a VMM be structured as to allow guest OSes to share hardware resources while balancing issues of performance, dependability, security, and software maintenance effort?

2. **Device-level extensions.** How can device-level system interfaces be extended to allow the safe introduction of new features with reasonable performance, while making extension development both reasonably easy and portable across OSes?

3. **Managing device access in VMM-based clusters.** As virtualization is deployed into large cluster environments, how can devices be managed in a facility-wide manner, while catering to new capabilities, such as migration, that are afforded by VMMs?

The first of these problems has been encountered and addressed in the course of the group's work on Xen. Our paper on safe hardware access [FHN+04] describes how the use of various techniques, in particular split device drivers, may be used to address the many challenges faced in providing device access in a VMM. The design and implementation of split drivers is presented in this chapter as necessary background to understanding the extension framework that I have designed and built. While split drives are work in collaboration with the group at Cambridge, the exposition in this thesis is of

considerably greater detail than has been published in the past, and serves as a contribution of this thesis in the same vein as the Xen history presented in the previous chapter.

This chapter describes how the split driver model may be extended to allow generic device extensibility, and then describes how extensions to individual devices on a single physical host may be aggregated across a collection of hosts to form *device services*, allowing the isolation of device provisioning and management within a cluster environment. At the end of this chapter, the reader should understand how the three problems described above are solved by my work. The remainder of the thesis aims to validate these mechanisms by describing a set of useful device extensions that have been built above them.

While these contributions stem from experience in the development of Xen, I will further argue that the ability to provide low-level device extensions has hitherto been impractical in conventional operating systems, and is a strong motivation for the broad deployment of VMMs on future systems. The next section provides a detailed argument for the benefit of device extensions and the challenges involved in providing them. The remainder of the chapter then presents the solutions to the three problems described above.

3.1 Extending Devices

At the core of this thesis is the notion of a low-level device extension. In general terms, an extension is a transformation of the behaviour of a device that exists behind the interface to the device that is presented to an operating system. As such, device drivers themselves, which map OS interfaces onto physical device hardware are one form of extension. However, the work in this thesis is primarily concerned with extensions that add new functionality to a device, as if that functionality were a part of the device itself. This composition of physical hardware and software extensions, shown in Figure 3.1, are described as *soft devices*. Soft devices preserve the interface presented to the OS, and so may be easily deployed in place of the raw device by nature of interposition. They may optionally present an out-of-band control interface, allowing the extension to be manipulated by external tools.

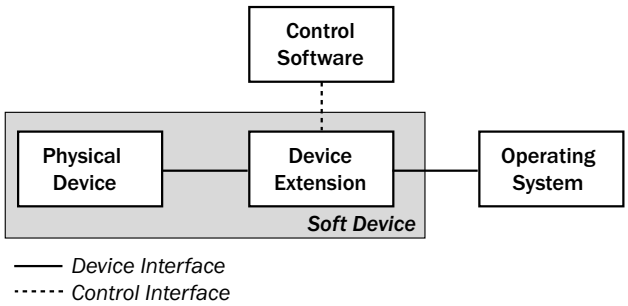


Figure 3.1: Soft Device Overview

The ability to easily construct device extensions is desirable to both hardware and software developers for similar reasons. OS developers may desire to provide new, low-level functionality that is not available in-hardware. By extending a disk, for instance to provide encryption, at or very near the device driver interface an extension may be applied across the file systems that are provided by the OS, while concurrently remaining applicable across the wide range of physical disk devices that may be used. Conversely, hardware designers may desire to prototype new functionality in software first, TCP offload for instance, to evaluate the benefits of new features and their impact on system software prior to beginning expensive hardware prototyping.

3.1.1 Performance and Safety

On conventional systems, providing device extensibility is a very challenging problem. As a subset of existing work into the more general problem of operating system extensibility, device extension code must strike a balance between requirements for both performance and safety: On one hand, extensions must be sufficiently low-overhead as to be realistically deployable, while on the other they must not represent a liability to the system in the case of crashes. The first requirement has traditionally meant attempting to move extensions into the kernel, where they do not incur overheads on protection domain crossings. Meanwhile, the second requirement has historically involved the exploration of mechanisms to ensure the safety of extensions that run in the same protection domain as privileged code, such as type-safe languages.

This thesis argues that VMMs provide a considerably more appropriate cut-

point for the introduction of extensions than do OSes because the interface to devices that is visible at the VMM level is considerably *narrower* that that of the OS. With regard the two issues just mentioned, performance and safety, the approach taken in this thesis will be to isolate extensions in a separate virtual machine. This extreme isolation of extensions results in what is likely the highest possible degree of safety that can be provided on conventional hardware: extensions are isolated from both crashes and resource over-consumption by merit of the VM sandbox. By beginning with strong isolation, this approach treats performance simply as a requirement of any practical solution, specifically that extensions must be "fast enough" to deploy in practise. The performance impact of VM isolation will be demonstrated later in this chapter to be sufficient to build deployable extensions for both disk and network devices. Rather than focusing on microbenchmarks, performance is characterized using complete, whole-extension benchmarks to demonstrate the ability to saturate both disk and network interfaces on modern server-class hardware. It is worth noting that while this is not a performance-centric thesis, that the results described here far exceed the published results of other recent VMM-based extension work [WCSG04], which is unable to saturate device hardware due to substantial extension and virtualization overheads.

3.1.2 Software Engineering

While concerns of performance and safety have dominated past research into the more general problem of OS extensibility, they are only a portion of the set of concerns that should be considered. In practical terms, modern systems have been at an impasse with regards the development of deployable extensions for quite some time. Considering device drivers as an extension (as described at the start of this section), developers must re-implement driver support for each OS that uses a piece of hardware, and often across versions of individual OSes. The difficulty of maintaining OS extensions at the device level has been described with respect to tracking the Linux kernel for the Nooks driver isolation work [FGCW05, SBL03]. The lack of uniformity, and in some cases availability, of driver interface across OSes has been described as a barrier to entry for new OSes and a considerable overhead to device vendors [UDI99]. Indeed, one of the major contributions of the Flux OSKit [FBB+97] was to provide a bridge to an existing pool of

drivers for new OS projects. This is a crucial point in the consideration of operating systems throughout recent history: Drivers themselves have not been portable across operating systems, and so a generalized notion of device extensions has obviously been impossible. A major claim of this work is that the structure of a VMM lends itself intrinsically to freeing OSes from the device-level ossification that has been present in systems over the past twenty years, and represents a major opportunity to enable innovation in low-level systems code.

The problem of supporting physical devices was encountered in the early versions of Xen in that, as with most previous systems, physical device drivers were a part of the VMM itself. The original version of Xen was based on the Linux kernel, and was left having to track changes to that source in order to support new emerging devices. This situation held for almost a year of development, and proved to be a tedious and time-consuming task. Xen tackled the problem by removing device support from the VMM altogether, opting instead to present raw device interfaces to "device VMs". This approach, which is described in the following section, allows the use of physical device drivers from *any* OS that runs on Xen, and frees the VMM from needing to track a specific OS's device support.

This is equally applicable for soft device-based extensions. A major concern in allowing the development of extensions is catering to maintainability: Extensions should remain useful over time, despite the fact that the OS code bases that they serve are both varied and rapidly evolving. The split device architecture described in the next section introduces narrow, idealized, device interfaces which form the basis of extension mechanisms. By interposing on these interfaces, extensions may be applied across any OS that is used on the VMM.

Besides maintenance, a second major issue in allowing the construction of device extensions is the interface available to the developer. Traditionally, such extensions are written directly within the source of the target OS. In addition to the maintenance concerns already discussed, this form of development imposes a great deal of initial overhead on the developer. Working within the source of an OS requires that the developer have a reasonably thorough understanding of the subsystem that they hope to extend, in addition to general programming idioms such as memory allocation and lock-

ing, which are specific to individual OSes. This developmental barrier to entry makes extension development unattractive, especially to novice developers, and results in code that is likely to destabilize a system. The extension mechanism described in Section 3.3 allows the development of extensions in user-space of the isolated extension VM. Developers are free to use whatever OS and development tools they desire to work with, and may also incorporate debuggers, which are often unavailable when developing within the OS. This is similar to the approach taken with user-level device drivers in other systems [EG04, Chu04, Hun97].

VMMs are a very coarse grained tool for the decomposition of systems software. They provide isolation at the level of an entire operating system, and only primitive communications mechanisms across VMs. The solutions described in the remainder of this chapter aim to demonstrate that in the case of devices this decomposition is a a useful one specifically because of the strong isolation that is provided between the component parts: Not only are extensions isolated from crashes and resource exhaustion that would traditionally render a system unstable, but their source code and administration are separated from that of both the devices and the OSes that they are applied to.

3.2 Split Drivers

As described above, including device support directly within the VMM presents two major problems. First, a driver base is typically borrowed from an existing operating system such as Linux, resulting in the software maintenance responsibility of tracking that source to ensure that drivers remain up-to-date. Second, as drivers are a major source of bugs resulting in system instability, their inclusion in the VMM itself destabilizes the platform as a whole. As the VMM forms the bottom-most software layer in a system, it is desirable to minimize its footprint in the interest of achieving some confidence of reliability and security. At the time of this writing, the Xen hypervisor for x86 is almost exactly 60K physical source lines (SLOC) of C and 1727 lines of assembler[1]. In contrast, the Linux 2.6.12 drivers

[1] All source measurements were gathered using SLOCCount, available at `http://www.dwheeler.com/sloccount/`.

subtree contains 2.1 million source lines of code. Not only is this a massive amount of code, it is also rapidly changing. A November 2000 analysis of the 2.2.16 kernel reported only 870,000 source lines of driver code, and a total kernel size (including drivers) of just over 1.5 Million SLOC [Whe00].

To address this issue, Xen uses an approach involving *split drivers*. Physical device drivers are removed from the VMM and run in isolated VMs, where faults may be contained and native OS drivers may be used. A pair of virtual device drivers is then used to allow client, or *front end* domains to have access to physical devices managed by the *back end* VM. The remainder of this section elaborates on the motivation for split device drivers and explains the details of their construction.

3.2.1 Isolating Driver Code

As of Xen 2.0, drivers were removed from the hypervisor and run in *driver VMs*. The VMM's responsibility was thus reduced to the simpler task of isolating access to device hardware to device VMs managing individual devices. On the x86, achieving this isolation involved the following general mechanisms:

- **I/O registers.** Memory mapped device registers are restricted to the managing device VM; Xen validates attempts to include device registers in a VM's page tables. The x86's I/O port space access bitmap is updated on context-switch into a VM to ensure that I/O port access instructions (e.g. `INB` and `OUTB`) may only be applied to managed devices.
- **Interrupt notification.** Xen retains control over the system's interrupt controller and provides a virtualized interrupt notification to the appropriate VM. This interrupt virtualization, called an *event channel* is described below.
- **Device configuration.** Xen virtualizes access to the generic PCI configuration space through which devices are detected and configured. VMs are able to see only the devices that they manage, and the hypervisor is able to validate configuration requests to ensure that they pertain to a permitted device.

This approach allows unmodified drivers to be hosted in individual driver VMs, where crashes may be isolated and drivers may be restarted upon failure without necessitating the restart of client VMs. A single weakness in the isolation provided by this model is that of device access to machine memory through DMA on modern systems: While drivers are prevented from accessing memory outside their own VM, they may program device DMA to read or write arbitrary memory within the system. This problem is a hardware weakness that is addressed by the availability of IOMMUs on modern chipsets; with the use of such MMUs, devices may be restricted to access a specific subset of the host's memory.

Practically speaking, this DMA weakness on current x86 hardware does not present a security or stability problem with regard to the interface presented to frontend VMs in the split driver model. Front ends access devices through narrow interfaces in which all addresses passed to devices are validated and typically translated; the DMA weakness is simply that malicious (or very badly broken) drivers in the backend VM could potentially compromise memory isolation in the system in the absence of an IOMMU.

3.2.2 VM Device Interface

With drivers themselves isolated in individual VMs, a second major design issue in the VMM is in deciding how to represent devices to VMs wishing to share access to a single device. While devices with a single client VM may be exported directly, many devices – storage and networking in particular – will be used by many VMs concurrently and the hardware interface may not be safely shared.

The problem of multiplexing access to device hardware has been a fundamental issue in the design of VMMs throughout their history, and there are two general solutions: hardware emulation, and idealized interfaces. In hardware emulation, a VM is presented with a virtual piece of hardware, typically a simple and common device for which drivers are broadly available. The VM runs the associated driver, and the VMM emulates this hardware, decoding operations and mapping requests down on to the physical device as appropriate. This approach is taken by VMware workstation [SVL01], where the desire to support a wide range of unmodified operating systems clearly supersedes the requirement for high-performance de-

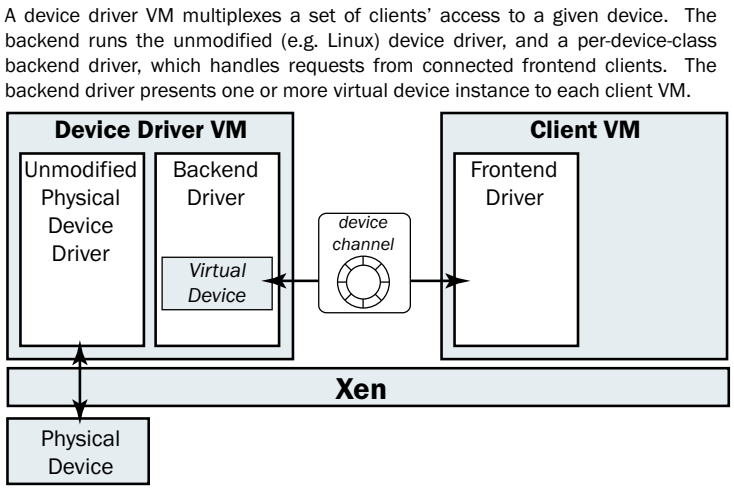


Figure 3.2: Split Driver Structure

vice access: As physical devices often involve several emulated I/O instructions per actual operation, this approach generally involves a reasonably high overhead.

A second approach to sharing device access is to present an idealized interface representing a class of device (e.g. a generic network interface) to the VM, and to write a new device driver specifically for this interface. This is the approach taken in Xen, and it has been used successfully in the past by other systems [BDGR97b, EFO95, WSG02].

Figure 3.2 shows the structure of split drivers that result from the combination of isolating physical drivers into driver VMs and exporting access to these devices over idealized device interfaces. Virtual devices are accessed over *device channels* – a combination of shared memory and event notification – between front end and back end VMs. The remainder of this section details the design and implementation of split drivers.

3.2.3 The Data Path: Device Channels and Grant Tables

Device channels are the general term used to describe communications between VMs above Xen. Xen does not attempt to abstract inter-VM communications by providing a specific inter-process communication (IPC) API. Instead, it provides the primitive mechanisms upon which arbitrary forms

of IPC may be composed. These primitives are broken into two components: **Notification** mechanisms, provided by the event channel interface, allow one VM to send a one-bit signal to another. **Data Transfer** mechanisms allow the sharing and exchange of memory pages between VMs. By building on top of these two sets of interfaces, VMs may construct arbitrary communications mechanisms, allowing complete flexibility in issues such as message batching and buffer sizes.

3.2.3.1 *Event Channels*

Event channels provide a notification mechanism between VMs— effectively a virtualized interrupt controller. Each VM has a bitmap of 1024 "ports", on which it may receive a one-bit notification. The API provides a pair of interfaces to bind ports between VMs, generating mappings in Xen that are represented by the tuple *(domain1, port1, domain2, port2)*. Once a channel has been established, either end may request that Xen generate a notification, which will result in the event bit being set for the receiving VM and a hint to Xen that it should be scheduled at some point in the future.

Aside from notification requests, Xen provides simple interfaces to manage event channels. A channel is created in one of two ways: Either a special privileged domain will request the creation of a channel between a pair of domains, in which case Xen will bind the channel and return the pair of newly assigned ports. Alternatively, a VM may request an unbound channel be created between itself and another domain. In this case, Xen will assign a local port to the unbound channel and at some time in the future the remote domain may request to connect to the *(domain, port)* pair, resulting in the completion of the binding.

Handling notifications is left to OS implementations, where they are generally bound to the interrupt subsystem. XenLinux provides mechanisms to assign interrupt handler functions to specific ports, allowing a convenient binding between event mechanisms and the existing code. The event interface also provides a matching bitmap to the notification ports on which it may mask inbound notifications. This provides a means for a VM to request that Xen not activate it on the arrival of new events to a given port.

3.2.3.2 *Shared Memory*

In order to allow data to be efficiently moved between domains, Xen provides a set of primitive operations to manage shared memory. The presence of a VMM adds an additional level of virtualization to memory in the system: We refer to the physical memory on a system as *machine* memory as it is addressed using the physical memory addresses on the machine. This memory is then subdivided and presented to a VM as a *pseudophysical* memory region, which the VM will treat as physical memory. These terms to describe memory in a VMM-based system are carried forward from Disco [BDGR97b].

Xen maintains a table mapping the assignment of machine pages to pseudophysical page addresses in each domain. Each machine frame on the host is either unallocated, or belongs to a specific domain. Domain pages are both typed and reference counted by Xen: typing ensures that Xen is aware of special-purpose (e.g. page table) pages and can validate them specially, while reference counting ensures that mapped pages remain available until all mappings in a VM are released.

Paravirtualized OSes achieve high-performance by working with both pseudophysical and machine page addresses. A VM may directly fill PTEs using machine frame numbers, allowing them to be used directly by the hardware after being validated by Xen. VMs typically take advantage of a physical-to-machine mapping table to translate between pseudophysical frame numbers, used by an OS to manage memory, and the machine frames used by Xen. The range of pseudophysical pages a host has represents the maximum amount of memory that may be assigned to that host. However, it is possible to reduce the memory that is actually assigned to a host using a *balloon driver* [Wal02]. Installed in a guest OS, the balloon driver provides a means to internally reserve an amount of memory, and then release the underlying machine pages back to Xen. This is a useful technique to resize the amount of memory available to a VM at runtime.

To share memory in Xen, it may be either *mapped* or *transferred* between domains. Memory may be mapped into a VM's pseudophysical address space by first releasing a range of pages using the balloon driver. To allow virtual memory mappings to these "foreign" pages, Xen provides a hypercall interface that allows machine addresses from other domains to be mapped

into a VM's page tables. These calls ensure that mappings are valid and permitted, and that pages are correctly reference counted.

In some situations, it is not appropriate to simply map memory belonging to a foreign domain. A primary example of this is in providing high-speed network communications, where we desire to deliver a received page directly to the intended VM without incurring an extra copy. As the destination is not known until the page has been received into memory and the packet headers have been examined, it is impossible to pre-map a destination page to receive the page into. Instead, VMs use the balloon interface to relinquish a set of pages to a free list, into which packets are received. Pages containing received packets are then transfered to the appropriate domain, the ownership of the page is transferred and the page is added to the VM's pseudophysical memory.

3.2.3.3 *Grant Tables*

The map and transfer facilities described above provide the necessary primitives for page sharing between VMs, but do not provide much flexibility in terms of managing the ability to share pages. Until Xen 3.0, the ability to modify memory mappings was a single privilege flag in a VM's domain structure – a VM could either make no foreign memory requests at all, or it could modify mappings belonging to any domain on the system.

Motivated primarily by the need to provide an isolated device driver interface, but also by the broader goal of designing general page-sharing mechanisms for VMs, grant tables were added as an extension to the existing memory mapping interface described above. Grant tables reflect a similar approach to the unbound event channel allocation scheme described in 3.2.3.1: A VM may "grant" access to a page of its memory to another domain, and at some point in the future, the remote domain may map that page into its own address space.

The grant table interface introduces the explicit notion of a *foreign mapping*: the ability of a VM to map a page that it does not own, given the owning VM's permission. To create a foreign mapping of a memory page, a VM must present a valid *grant reference* to the hypervisor in lieu of the page number. This reference comprises the identity of the granting VM, and an index into that VM's *grant table*. Every VM owns a private grant

table that it shares only with the hypervisor, in which each entry is a tuple $(\mathit{map}, V, P, R, U)$ permitting VM V to map page P into its address space; asserting the boolean flag R restricts V to read-only mappings. The flag U is written by the hypervisor to indicate whether V currently maps P (i.e., whether the grant tuple is *in use*).

When the hypervisor is presented with a grant reference (A, G) by a VM B, it first searches for index G in VM A's *active grant table* (AGT), a private table that is only accessible by the hypervisor. If no match is found, it reads the appropriate tuple from the guest's grant table and checks that T=*map* and V=B, and that R=*false* if B is requesting a writable mapping. Only if the validation checks are successful will the hypervisor copy the tuple into the AGT and mark the grant tuple as in use. The AGT is a private, in-hypervisor copy of the VM's grant table that also includes additional book-keeping data such as mapping reference counts. Since it cannot be accessed by a malicious or buggy VM, the hypervisor can guarantee the integrity of this critical data.

The hypervisor tracks uses of grant references by associating a usage count with each AGT entry. Whenever a foreign mapping is created with reference to an existing AGT entry, the hypervisor increments that entry's count. The grant reference cannot be reallocated or reused by the granting VM until the foreign VM destroys all mappings that were created with reference to it.

The grant table interface supports transfer operations similarly: A transfer tuple is of the form $(\mathit{transfer}, V, P)$, permitting VM V to transfer ownership of one of its pages to the specified VM. A transfer tuple permits the transfer of a single page.

3.2.3.4 *Communication Rings*

The current split drivers structure each device channel as a bi-directional ring buffer with two pairs of producer/consumer pointers. A balanced number of request (client to server) and response (server to client) messages are passed back and forth on the ring.

As shown in Figure 3.3, a ring is partitioned into request and a response queues, and domains only work within their own space. This can be thought of as a double producer-consumer ring – the ring is described by four pointers into a circular buffer of fixed-size records. Pointers may only advance,

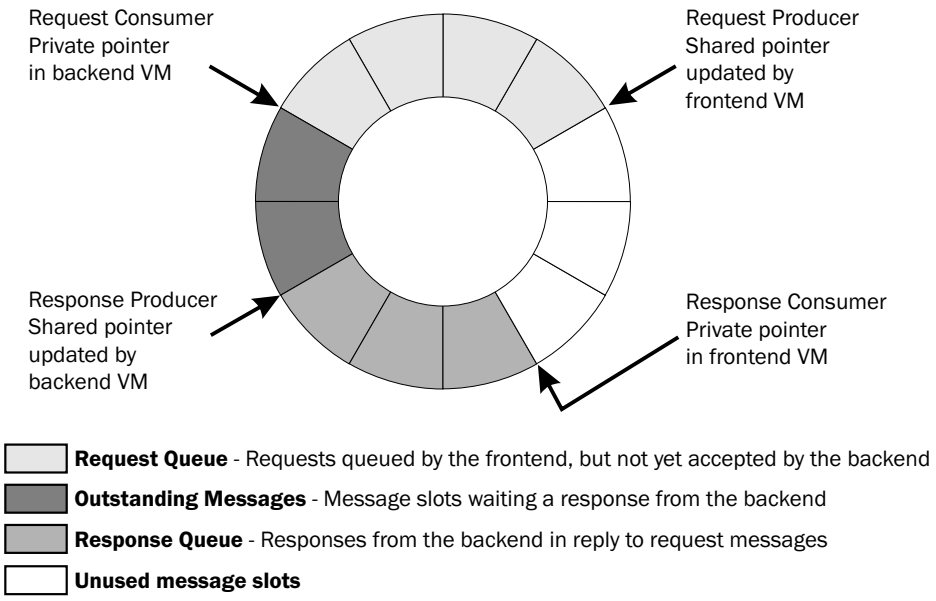


Figure 3.3: Shared-Memory Ring

and may not pass one another.

By adopting the convention that every request will receive a response, not all four pointers need be shared and flow control on the ring becomes very easy to manage. Each domain maintains its own consumer pointer, and the two producer pointers are visible to both. As shown in Figure 3.3, request and response message regions chase each other around the ring.

To ensure safety in accessing the shared memory region, drivers keep private copies of in-use ring data. This generally includes retaining copies of requests that are being processed, and private versions of producer pointers. This "defensive" approach to managing the shared data structure provides stability in the case of a misbehaved or malicious driver on the other side of the ring, and furthermore enables recovery after driver failure or VM migration.

Bi-directional rings can trivially be converted into uni-directional rings by using only a single pair of producer/consumer indices. Unbalanced communications, for instance console I/O, may be implemented using a pair of uni-directional rings – one in each direction.

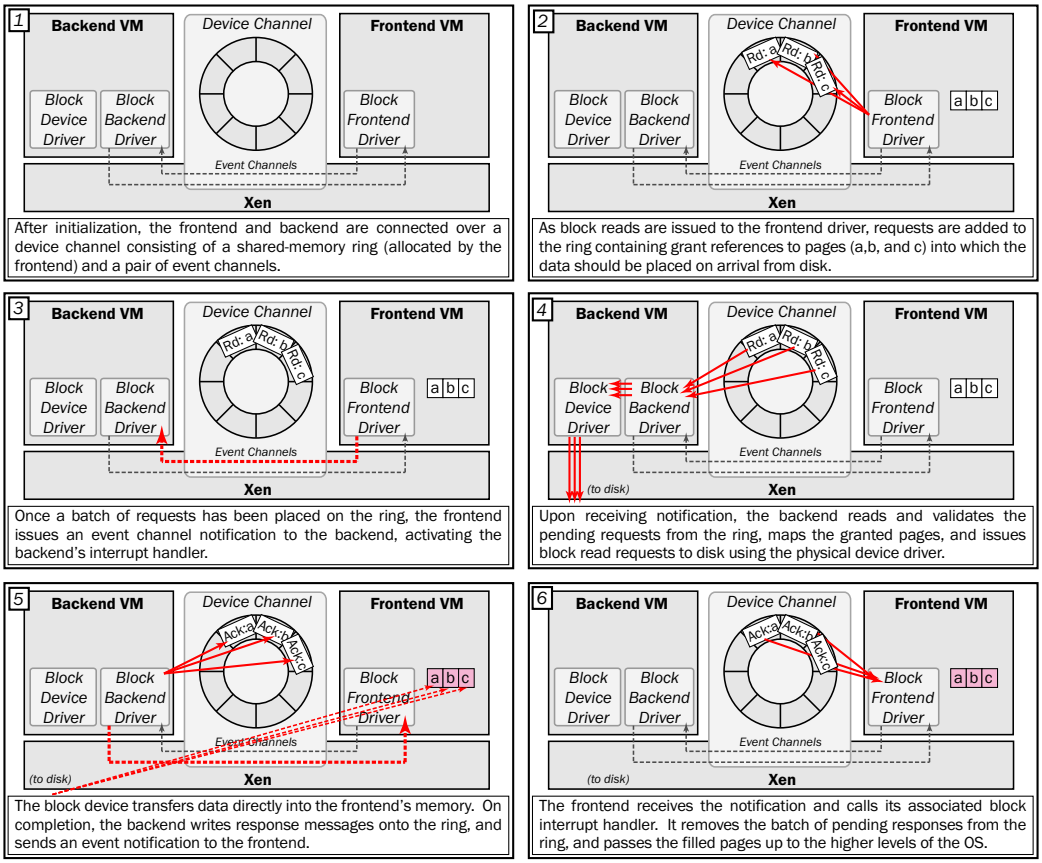


Figure 3.4: Device Channel Example: Block Read

3.2.3.5 *Device Channel Example: Block Read*

Figure 3.4 shows a high-level illustration of how device channels are used to issue a block read request across a split driver. Once the device channel has been initialized, the frontend places a set of read requests on to the shared ring. Request messages placed on the ring are described in more detail later in this section; they contain details such as the virtual block address to read and grant references for the pages of memory into which the inbound data should be placed. Requests often arrive in batches, both due to the nature of block workloads and the existence of OS prefetching techniques. Once a batch of requests has been placed onto the ring, the frontend will send a notification on the event channel to trigger activation of the backend driver.

The event channel maps to an interrupt handler in the backend driver. When the notification is received, the backend will remove pending entries on the

ring, advancing its request consumer pointer as described above. Each request is validated, the grant references are mapped into the backend VM, and requests are issued to the physical device driver.

As references to the machine pages to read into are included in the request as grant references, the data is transfered directly to the frontend's memory. The backend is notified as read operations complete, at which point it writes response messages onto the shared ring and then notifies the frontend that requests have completed.

Block writes work almost identically to reads, except that data moves from, instead of to, frontend memory. The mechanism of passing request messages on shared rings with references to associated data pages and using event channel-based notification to achieve batching are common to other split drivers. As described later, the network drivers use grant transfer as opposed to mapping operations.

3.2.4 The Control Path: Control Interfaces and XenStore

The control interfaces in Xen have evolved constantly throughout its development. In complement to the device channels, the control interfaces serve to provide a non-performance-critical, general API to allow management tools and VM-based OSes to share control information. The control interfaces are used to perform tasks like device setup, which is generally a two phase process: First, the control tools indicate to a backend driver what virtual devices it should export to a new VM. Second, the VM boots and probes its associated backends for available devices, and negotiates device channel-based connections to them.

3.2.4.1 *XenStore*

Xen unifies VM configuration data in a persistent store. *XenStore* provides a hierarchical name space within which configuration data can be exchanged. The store provides a persistent view of the configuration state of the system, and a point of indirection through which many clients may interact with regard to a common set of configuration interest. This approach is similar to shared whiteboards as used in distributed computing, and as such is intended to scale out beyond a single host to allow configuration management for a VM clusters. XenLinux includes a XenBus driver, which allows the

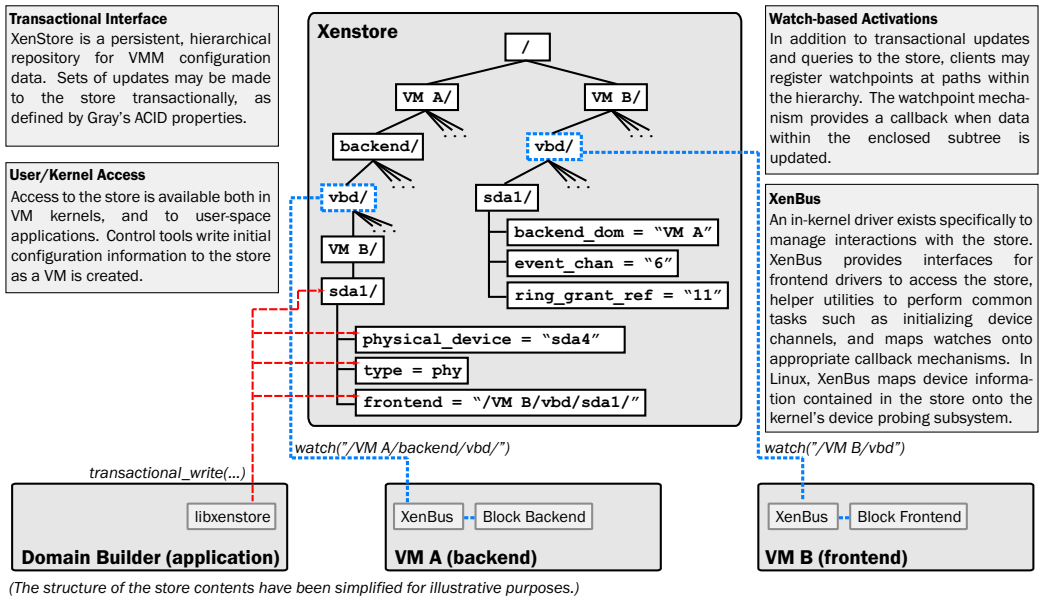


Figure 3.5: XenStore Overview

mapping of portions of the store onto the Linux device enumeration mechanisms. In this manner, virtual device configurations may be written into the store, and trigger registration in a guest VM as if a device had been hotplugged into the system.

XenStore's hierarchical name space bears a strong similarity to a traditional file system. Each domain is represented by a subdirectory in the store, and aspects of a domain's configuration are placed in subdirectories within that domain's tree. Values in the store behave much like files in a traditional file system, but generally contain human-readable text. The store supports transactional updates, so that sets of new name-value bindings may be applied all at once, or not at all.

Communication through the store is achieved by registering *watch points* on subtrees in the hierarchy, essentially a subscription to the region of interest. After a watch has been registered, changes to the associated subtree will fire the watch, and indicate the value within the subtree that has changed.

Figure 3.5 provides a simplified high-level view of the structure of XenStore. The figure shows configuration information relating to block devices, and the clients involved in block device connection: The domain builder, which

is a component of the Xen control tool set; and a frontend and backend VM that will form a split driver-based connection for a block device. The diagram shows a subset of the control information and mechanisms involved in device set-up, and is intended to provide an overview of this process. A detailed account of block device setup is provided later in this section.

Device configuration and initialization involves collaboration between the three entities mentioned above, using the store to share configuration data. As a new domain is created, the domain builder tool, which runs in user-space in domain 0, writes configuration information describing the new VM into the store. In addition, it writes device configuration information into the associated backend domains' device subtrees. In the figure, the domain builder performs a transactional update, adding a set of configuration data regarding a new virtual disk to be served to *VM B*. On commit, this data appears in the store, and results in the activation of a watch that the backend domain, *VM A*, has registered. The backend driver will validate the new disk request, and prepare for a connection from the frontend. Part of this preparation will involve the addition of a second watch point, on *VM A*'s device subtree, as specified by the `frontend` path.

When the frontend VM boots, the block driver will initiate the connection of a new device channel. The control tools will have provided the `backend_dom` id, which the frontend will use to allocate an unbound event channel, and a grant reference for the ring page. These entries are then written to its directory in the store. The update will trigger the watch point that was added by the backend, which will then connect to the device channel, completing the initialization of the split driver.

Interactions with the store from user space are provided through `libxenstore`, a library of accessor functions into the store. In-kernel interactions are managed through the *XenBus* device driver. XenBus provides accessor functions for the store and helper functions for common tasks such as the initialization of device channels. In addition, it maps a VM's device details as represented in the store onto the OS's device probing and hotplug code.

As mentioned, the block device description later in this section will provide a more detailed overview of the use of XenStore and XenBus to initialize devices.

read(*vblock*) → *page*	Read the data stored at virtual block address *vblock*, placing it in the specified page.
write(*vblock, page*)	Write the contents of *page* at the specified virtual block address.

Table 3.1: Xen's Virtual Block Device Interface.

3.2.5 The Virtual Block Interface

The previous sections have explained the underlying mechanisms used in the composition of split drivers. To illustrate how these are used in practise, we now consider the implementation of the virtual block interface.

3.2.5.1 *Data Path*

The device channel for virtual block devices in Xen makes use of a very simple protocol for block requests. The channel is a symmetric shared memory ring (as described above) where a client VM issues requests, and a backend returns responses as the requests are completed. There is a one-to-one relationship between requests and responses, although the backend is free to reorder responses. The protocol currently consists of two simple request messages, shown at a high level in table 3.1.

Figure 3.6 shows the C structure that describes a block request to detail the *vblock* and *page* parameters shown in the table. A virtual block address is described simply as a 64-bit sector offset from the start of the virtual device. An extent of data to read or to be written is described as an array of page-sized segments associated with the request. Each entry in the `frame_and_sects` array describes a page in the client VM where data should be read from or written to, and a range of sectors within that page to operate on. Modern file systems generally interact with the disk at page rather than sector granularities, but this interface ensures sector-granularity access remains available when required.

Data pages are accessed using the grant *map* interface: The backend domain releases a region of its physical memory, creating a hole in its own physical address space. The size of this physical region is equal to the maximum number of segments per request multiplied by the size of the request ring, allowing a completely full block request ring to be mapped into the backend's memory. The default Xen configuration uses a 64-entry shared ring,

```
#define BLKIF_OP_READ      0
#define BLKIF_OP_WRITE     1

#define blkif_vdev_t   u16
#define blkif_sector_t u64

typedef struct blkif_request {
    u8             operation;     /* BLKIF_OP_{READ|WRITE}                */
    u8             nr_segments;   /* number of segments                   */
    blkif_vdev_t   handle;        /* the device this request pertains to  */
    unsigned long  id;            /* private guest value, echoed in resp  */
    blkif_sector_t sector_number; /* start sector idx on virtual disk     */

    /* @f_a_s[4:0] = last_sect ; @f_a_s[9:5]=first_sect                    */
    /* @f_a_s[:16] = grant reference (16 bits)                             */
    /* @first_sect: first sector in frame to transfer (inclusive).         */
    /* @last_sect:  last sector in frame to transfer (inclusive).          */
    unsigned long  frame_and_sects[BLKIF_MAX_SEGMENTS_PER_REQUEST];
} blkif_request_t;
```

Figure 3.6: The Block Request Structure

with a maximum of 11 segments per request. These numbers are selected largely out of a desire to maximize bandwidth, while allowing the shared ring to fit on a single 4KB page.

As requests arrive in the backend driver they are validated and their data pages are mapped into the empty pseudophysical address space. Requests are then translated into real block I/O requests and issued to the block subsystem. In the Linux backend, these requests are asynchronous and result in a callback on completion. The callback handler fills out a response message, and places this on the shared ring. It then uses an event channel notification to indicate to the client VM that the request has completed.

3.2.5.2 *Control Path*

Control operations relating to virtual block devices include adding and removing block devices for a client VM, negotiating the setup of shared-memory device channels, and interrogating device metadata such as a virtual disk's capacity and sector size.

As mentioned above, the configuration and control of virtual disks is managed through XenStore. When a virtual disk is created, the control tools will create a new directory in the backend's block device directory. This directory is typically named using a unique identifier of the device being exported, for instance the major and minor numbers of an exported physical

disk (represented as UNIX device names in Figure 3.5 for readability). From this point, device setup proceeds as follows:

1. **Tools write details to disk to export in VMs' device trees.** Inside the directory, the tools will add entries describing the domain id of the client VM, and a path to that VM's frontend directory in the store. Entries are also added describing the physical disk or partition to export, and whether it is to be exported writable or read-only. Committing this update will fire the backend driver's watch point, resulting in it scanning the new configuration data. In addition to writing the backend's tree, the tools add a subdirectory for the new virtual disk in the frontends tree, including the ID of the backend VM.

2. **The backend prepares to connect the device.** On receiving the notification the backend validates that the device is valid and instantiates data structures to handle the new connection. It registers watch points on both the new backend device directory, and the frontend's directory (which it has been granted permission to), and waits for the frontend to start.

3. **The frontend VM boots, and initializes the device.** The frontend allocates the components of a device channel, writing details of an unbound event channel and a grant reference for the shared ring page into its device subdirectory. The frontend driver adds a watch the virtual disk subdirectory.

4. **The backend connects to the device channel.** The event channel is bound and the ring page is mapped into the backend. The backend then writes details of the virtual disk, including disk capacity, sector size, and additional device-specific details.

5. **The frontend completes the connection set-up.** The frontend receives the device info and completes the connection process, registering the new device with the OS. At this point, the disk is available for access.

Device removal is handled by the tools simply by removing the frontend directory. Both drivers take this as an indication that the device is no longer available, disconnecting the device channel and removing outstanding state.

3.2.6 The Virtual Network Interface

The virtual network interface is similar in nature to the block interface. The network interface is more complex in two significant ways, both of which relate to the asynchrony of network traffic: First, as the numbers of packets transmitted and received are not exactly symmetric, a single shared ring is not sufficient as a device channel for network traffic. Instead, two rings are employed, one for transmission and one for receipt of packets. Second, the destination of an inbound network packet is not known until *after* it has been written to memory and its protocol headers have been examined. This makes it impossible to write received packets directly into memory already belonging to the recipient VM.

To address this second issue, the grant *transfer* operation is used. The frontend network driver relinquishes a set of pseudophysical pages to the backend, where they are added to a free list. As inbound network packets are received, they are placed into pages on this list and protocol headers are examined. Once the destination VM is ascertained, the packet-containing page is transfered into the empty pseudophysical memory region provided by the VM. If a VM is unable to provide sufficient free pages to accept inbound packets, the backend simply drops the excess packets, returning the page to the free list.

Linux has an established history of being used to build PC-based routing, NAT, firewall, and related packet processing hosts for small networks. The network subsystem of the OS is very feature-rich, providing support for filtering, limited processing, and forwarding of packets at all layers of the protocol stack. The current backend driver simply maps VMs onto virtual interfaces as provided by the Linux network subsystem, allowing the packet processing facilities in Linux to be used to make packet forwarding decisions. The in-hypervisor virtual firewall router [WHTP02] is no longer used.

Typical Xen installations generally use one of two forwarding techniques within the backend VM: The Linux bridge tools allow forwarding of packets at the Ethernet frame level, and include support for MAC-address filtering and proxy ARP. With a bridged interface, VMs appear as additional hosts on the local link. Alternatively, the backend VM may be configured to use the Linux routing tools, acting as an IP-level gateway for the client VMs.

The bridging and routing extensions have comparable performance, and both allow the use of additional network forwarding tools, such as traffic shaping. The bridging tools are perhaps slightly easier to configure, while the routing path is undoubtedly better tested given the large number of home routers based on Linux.

3.2.7 Performance of Split Drivers

Figure 3.7 demonstrates the overhead of split drivers on a set of system benchmarks. All measurements in the graph were carried out on a Dell PowerEdge 2650 dual-processor 3.06GHz Intel Xeon server with 1GB of RAM, two Broadcom Tigon 3 Gigabit Ethernet cards, and an Adaptec AIC-7899 Ultra160 SCSI controller. The SCSI controller hosted two Fujitsu MAP3735NC 73GB 10K RPM SCSI disks. All tests used RedHat Linux 9.0 with the 2.4.26 kernel.

Using a native Linux 2.4.26-SMP kernel installation as a baseline (*L*), the graph shows two relative performance figures for each benchmark. The first measurement evaluates the overhead of virtualizing device access into the backend VM (Xen0): this evaluates just the isolation mechanisms in Xen for virtualizing device registers and interrupts. In the *Xen0* tests, the server is run directly in the driver VM. The second relative result (XenU) measures complete isolation using a split driver to isolate the benchmark application in its own VM. This adds the additional overhead of routing device requests over a device channel to the backend VM. In all cases, the benchmarked OS is configured to use 512MB of memory. In the *XenU* results, the driver domain has 256MB of memory, but does not use the buffer cache. Configuring all benchmarked VMs was intended to to ensure that the Linux instances being tested were as identical as possible.

The set of benchmarks is as follows:

- *Linux build time* - The time to build a Linux 2.4.26 kernel stored on the local ext3 file system.
- *Postmark (PM)* - A filesystem benchmark developed by Network Appliance which emulates the workload of a mail server. It starts by creating a set of 2,000 files with sizes of between 500B and 1MB each, and then performs 10,000 transactions on them. Each transaction is

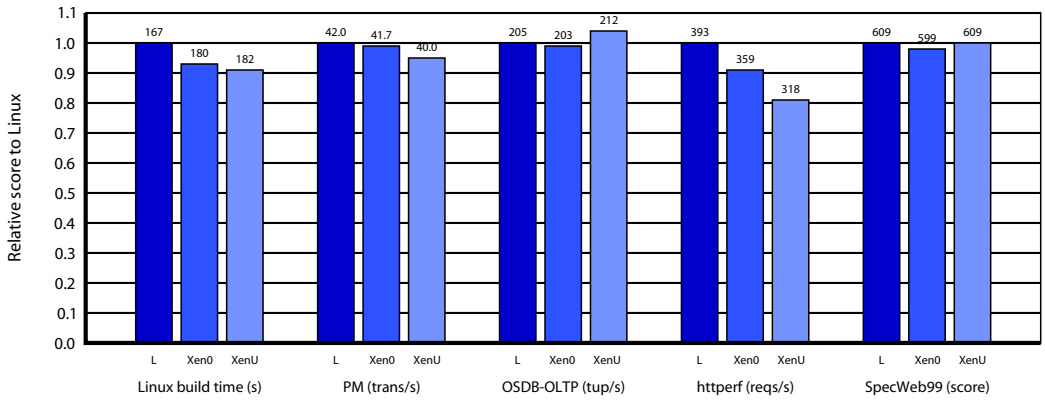


Figure 3.7: System Benchmarks of Split Drivers

composed of a set of file operations including creation, deletion, and appending of data. In total, over 7GB of data is transfered to and from the disk.

- *OSDB-OLTP* - An open source database benchmark that creates and populates a database and then runs a battery of queries and updates against it. The database used is PostgreSQL 7.3.2, and the benchmark is configured to use a 400MB database.
- *httperf-08* - A workload generator for web server benchmarking. In this case, httperf was configured to evaluate server latency, by allowing only a single outstanding request at a time. As such, this is effectively a ping-pong test for a single 64KB static web page, served using Apache 2.0.40.
- *SPEC WEB99* - An industry standard benchmark for evaluating web servers and the systems that host them. The test workload involves a mix of request types: 30% involve the generation of dynamic content, 16% are HTTP POST operations, and 0.5% execute a CGI script. Disk activity in this benchmark involves a 2.7 GB dataset of served data, and the generation of access and POST logs.

With the exception of *httperf* split drivers are within 10% of native performance. The *httperf* result illustrates the main weakness of the split driver approach: As device requests are routed across multiple isolated protection domains, there is a per-request overhead in terms of latency. As the *httperf* example allows only a single outstanding request, it is not network bound.

The test achieves a maximum bandwidth of approximately 200Mbit/s, and is limited on the end-to-end latency of request processing. In general, this is not a problem for server workloads, as request pipelining and batching allows latency to be amortized across requests and achieves a high throughput, as shown on the other benchmarks. A lesson from the *httperf* result is that this approach to isolation is costly for certain specific workloads such as fine-grained cluster computing, where synchronous dispatch is used to issue small pieces of computational work to hosts in a cluster. This latency overhead is a result of a design decision in Xen to optimize for the more common requirement of high throughput: the latency overhead is even more costly in the next section on device extensions as an additional stage is introduced to the pipeline; throughput, however, remains very high.

Figure 3.7 shows an interesting result for *OSDB-OLTP*, in which the split drivers actually outperform native Linux. This is a result that has recurred in many situations while investigating virtual block I/O for Linux. The virtual memory system in Linux is rather complex with regards the handling of dirty pages. Experience shows that the introduction of a request pipeline at the block interface tends to stabilize this behaviour, and often improves performance. This result is repeated in the next section with regard to soft devices for block I/O.

In concluding the presentation of the split driver approach, it is worth reiterating that split drivers achieve three main benefits in a VMM-based system:

1. **Drivers are isolated against failure.** Driver code executes in an isolated VM, and is given direct access to hardware. A thin virtualization layer in the hypervisor virtualizes and/or validates access to sensitive resources such as interrupt dispatch, I/O register access, and device configuration. Using this approach, driver failure is isolated, and driver VMs may be rebooted in the presence of failure and returned to a stable state without affecting the frontend VM. Most importantly, by removing drivers from the VMM, the system as a whole is more stable.

2. **Unmodified device drivers may be used.** By allowing direct hardware access to entire OS units, driver VMs may capitalize on existing driver source. Any OS that runs on the VMM may be used to host device drivers, freeing the VMM from the need to track changes to driver

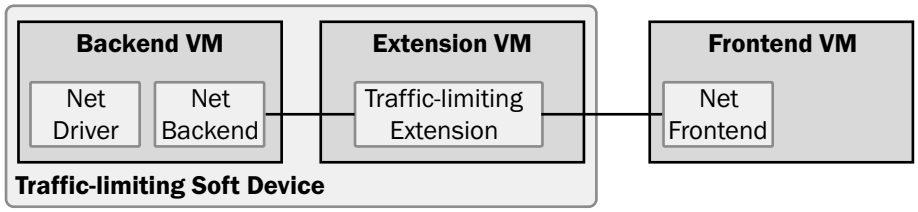


Figure 3.8: High-level View of a Traffic-limiting Soft Device

code in an existing OS code base.

3. **Devices may be shared across a variety of OSes.** Split drivers export access to physical devices across device channels by presenting an idealized interface to a class of device. This idealized interface is easy to write a new frontend driver for, allowing new OSes to be rapidly modified to use virtualized devices within the VMM.

In addition to these three benefits, the introduction of the narrow, idealized interfaces that are used on device channels facilitates the interposition of device extensions. The next section extends the split driver approach by allowing the introduction of new extensions to devices independent of the frontend OS, and backend driver.

3.3 Soft Devices

As stated in the introduction of this chapter, a major problem in the design of systems is that of determining how to allow the safe introduction of new software-based features to devices with reasonable performance, while making this extension development both reasonably easy and portable across OSes. By extending the device channel interfaces used by split drivers, extensions may be interposed between the front and backend drivers, examining and modifying device requests as they are issued.

As an example of such an extension, consider a modified network interface that includes traffic shaping and rate-limiting functionalities, which make it difficult for malicious software to generate certain forms of attack traffic. This soft device is shown at a very high-level in Figure 3.8. By implementing

this functionality in an extension interposed on the virtual network device interface, several benefits are achieved:

1. **Security.** As the new traffic limiting feature is intended to limit malicious behaviour, it should not be included directly within the host OS, which may be compromised and modified.

2. **Proximity.** Conversely, by placing the extension as a portion of the virtual network device, a tight coupling is maintained. In this example, the extension need not *drop* offending traffic, as it would were it added outside the physical host as a network device. Instead, it may simply opt not to dequeue packets from the transmit queue, deferring queuing overhead onto the VM itself.

3. **Portability.** As mentioned previously, implementing the extension outside the context of both the frontend and backend VMs allows source independence from both of these implementations. Extensions may be applied to any OS on the VMM which accesses the network using the split driver. Moreover, as explained later in this section, extensions may be written in userspace, allowing complete freedom of development environment.

4. **Isolation.** Finally, as extensions may be written as user-space applications, and may further optionally run in their own virtual machine, their failure is isolated from affecting the rest of the system. Extensions may easily be shut down and restarted while serving requests from running VMs.

The example of limiting malicious traffic within an extension is presented in considerably more detail in Chapter 4. Tens of such device extensions have been written over the course of this thesis; other examples include network address translation to share the port space of a single IP address across a set of VMs, copy-on-write and encrypted disks, and remote device access.

The remainder of this section discusses how a *device tap* may be used to interpose on the device channel used by split devices. It then goes on to present tap-based extension interfaces for both block and network devices.

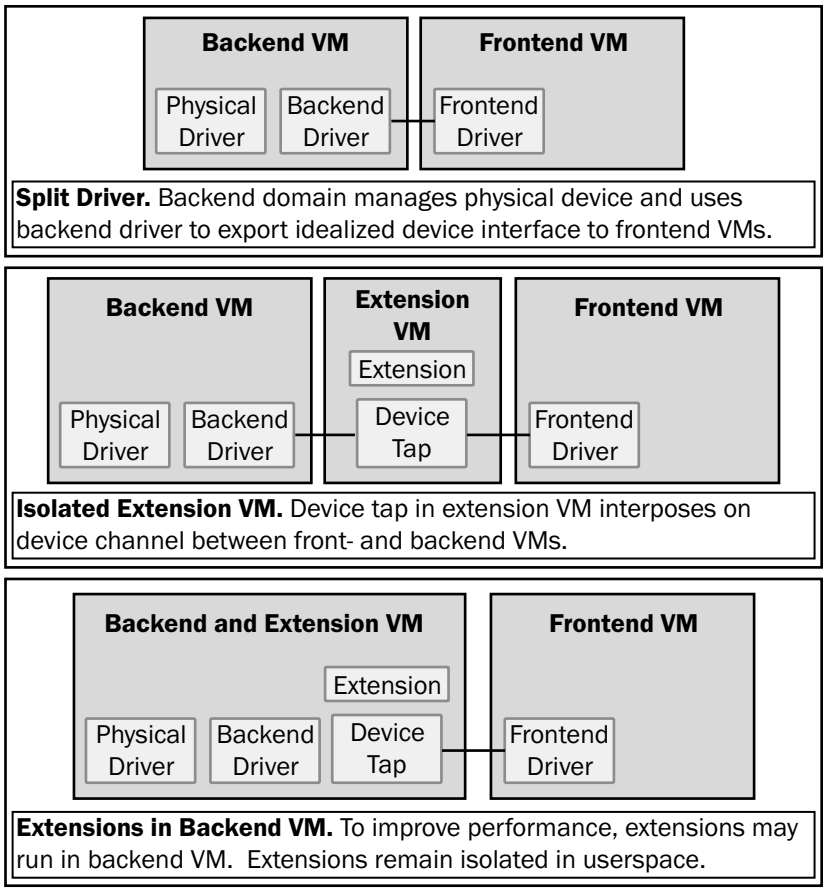


Figure 3.9: Overview of Device Tap Configurations

3.3.1 The Device Tap

The device tap is a device interface-specific driver that provides a means to extend devices in arbitrary ways by interposing on the device channel between the frontend and backend VMs. For example: developers may desire to simply trace request traffic in order to monitor usage patterns, they may wish to modify in-flight requests, or they may desire to construct a terminating device, which does not forward requests to a backend driver at all.

In order to accommodate these varied modes of operation, the device tap acts as a request switch for device channels. As shown in Figure 3.9, the driver may be plumbed into the device channel between the frontend and backend domains, and may operate on all requests that are issued. The tap

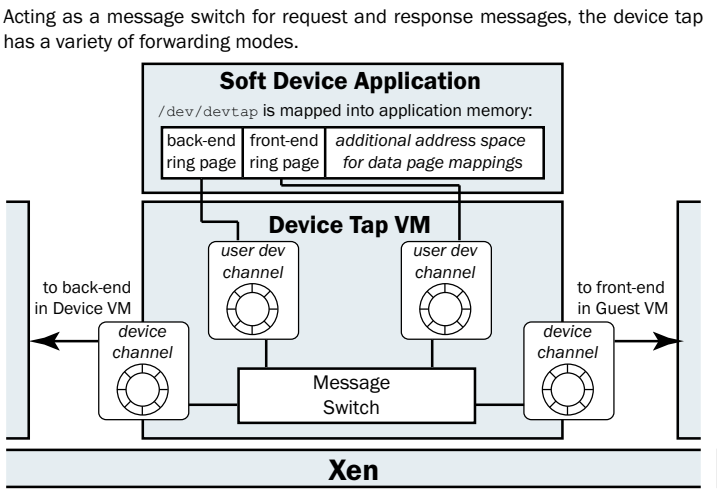


Figure 3.10: Device Tap Structure

driver is specific to the idealized device interface being used, but may be placed in either a completely isolated VM, or alongside the backend driver in the backend VM. Isolating extensions in a complete VM allows them the freedom to be implemented in any OS, using whatever development tools are desired. However, this approach incurs an additional overhead in terms of context-switching. As shown in Figure 3.9, if extensions are written for the same OS as the backend and physical device driver, the tap may be installed in the backend domain, and performance is improved. As extensions are generally written as user-level applications, this approach still maintains a reasonably high level of isolation.

Figure 3.10, shows a more detailed view of the device tap itself. The single device channel between the front- and backend drivers in the split driver configuration is broken into two rings. The tap interposes on this channel, appearing as a backend to the front and a frontend to the back. The device channels are then presented to applications in userspace using shared memory rings, exported over a character device that may be mapped into application memory. Where event channels were used for inter-VM notification on device channels, `poll()` and `ioctl()` syscalls achieve notification to and from extension applications and preserve batching of messages.

3.3.1.1 *Switching Modes*

The device tap itself acts as a switch for device messages. Once installed in the device channel, it may be configured for a particular switching mode, and will forward messages accordingly. Figure 3.11 summarizes three common switching modes used by the tap.

`MODE_PASSTHROUGH` is the lowest-overhead switching configuration. In this mode, messages are passed straight through the driver on to the opposite ring, and completely bypass the user rings. Passthrough can be used to implement kernel-level monitoring of block requests, or to implement soft devices in-kernel for improved performance.

`MODE_INTERPOSE` routes all requests and replies across the user rings. An application must attach to the device tap interface and pass messages across the two rings, allowing complete monitoring and modification of the request stream at the application level. This mode can be used to modify in-flight requests, for instance to build a compressed or encrypted block store. This is the mode of operation that is used for the majority of examples described in the later chapters of this thesis.

`MODE_INTERCEPT_FE` uses only the front-end rings on the driver, disabling the back-end altogether. This mode allows the development of new device functionality completely in user space of an isolated virtual machine. This mode is used the the Parallax storage system, which acts as a user-space backend and then sends requests to the file system or network using the traditional OS interfaces.

In addition to passthrough and interposition modes, the original tap implementation provided support for applications to request copies of data, with the intention that read-only access to data be made possible out-of-band, without placing the extension code directly on the data path. For most applications this proved to be impractical: incurring a fixed copy on the data path is expensive, and a copy-on-write mechanism for interdomain pages does not currently exist in Xen. In practice, placing hooks on the data path is sufficiently low overhead that this support has been removed as extraneous.

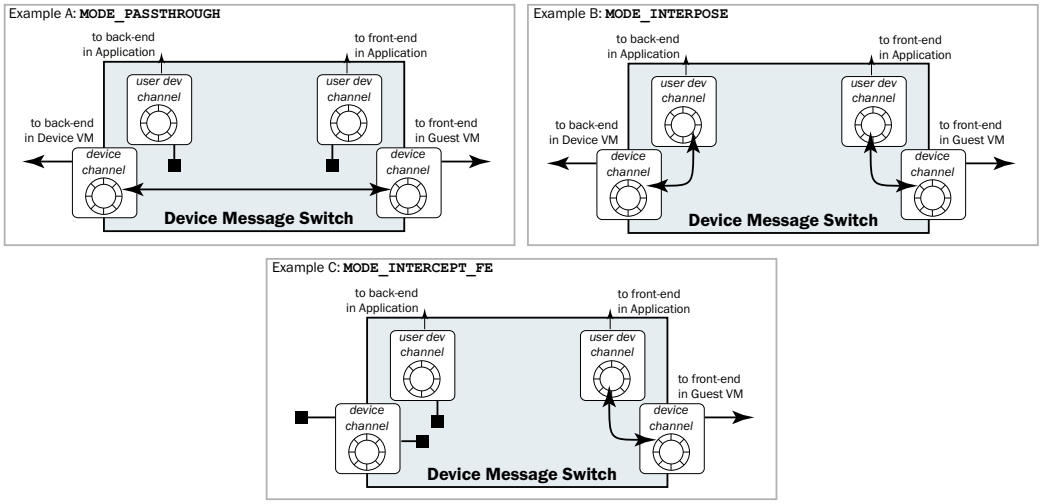


Figure 3.11: Examples of Forwarding Modes

3.3.1.2 *User Interface*

As mentioned above, the device channels between the tap and the front- and backend VMs are presented to userspace using memory mapping and system calls for notification. As with the existing split drivers, this approach avoids copy overheads and preserves the batching of messages on the device channels. For the device tap implementations discussed later in this section, additional support has been provided in the form of library interfaces for extensions. Libraries attach to the tap driver interface and provide data structures representing the front-end connections from attached VMs. Extensions are able to attach to individual devices, and register to receive their device requests.

An interesting avenue of exploration which has been left as future work is allowing extensions to be installed directly into the kernel. One approach of interest is that taken by the Click modular router [KMC+00], in which routing extensions use a common API in both user and kernel mode. This "develop in user, deploy in kernel" is an attractive approach to building extensions that could be incorporated into tap drivers in the future.

The remainder of this section discusses device taps for block and network devices that have been implemented as part of this work.

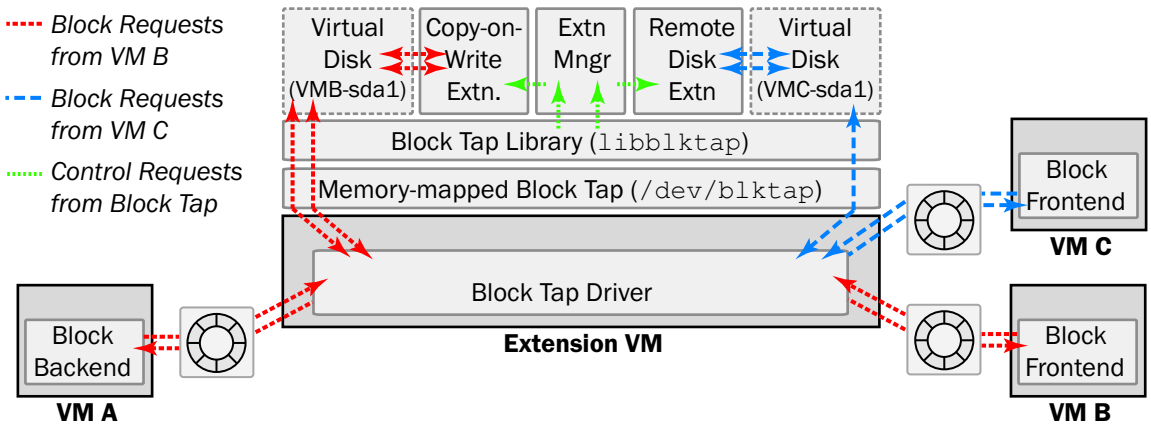


Figure 3.12: Structure of the Block Tap

3.3.2 The Block Tap

The block tap is a tap driver that allows the extension of block devices. It was the first tap driver to be implemented and was initially described in a paper at USENIX 2005 [WFHD05]. The tap was originally implemented to support the development of Parallax, which is described in detail in Chapter 5.

3.3.2.1 *Structure of the Block Tap*

The overall structure of the block tap is shown in Figure 3.12. This figure shows two extensions written to the library interface provided by the tap. The library generates control messages to an extension manager as client VMs add and remove virtual block devices. Based on watch points in XenStore, these messages may be used to trigger the instantiation of extension code to handle requests for the associated virtual disk.

The figure shows two extensions, each serving block requests from a different VM. VM B is provided with a copy-on-write disk extension which remaps writes onto an alternate backing store, rendering the disk served by the backend as immutable. VM C is served by an extension which forwards block requests to a remote host over the network, where they are issued to a backend driver. The network connections are not shown in the figure for clarity.

Note that the front- and backend VMs all connect over existing block device channel interfaces, and are unaware of the existence of the tap or the extension.

Technical Details of Block Tap Implementation

The block tap is implemented as a Linux 2.6 device driver, and has been maintained in the Xen tree for almost two years. Over the course of its development, several interesting technical challenges have been addressed in order to allow user-level extensions to achieve high-performance.

The most notable of these challenges has been in extending the Linux memory management system to handle the presence of pages of memory that do not belong to the virtual host. As a non-virtualized OS, it is understandable that this functionality is not a part of Linux; it is, however, crucial for high-performance.

When writing user-level extensions that access hardware managed by the extension VM, it is desirable to allow direct device access on pages of memory that have been mapped by the block tap from a client VM. While many modern operating systems, including Linux, have facilities to issue direct device requests (eliminating in-OS copies), memory mapped from a foreign VM is not part of the extension VM's pseudophysical memory, and as such the OS is unable to generate the appropriate machine address to program device DMA.

To solve this, the Linux memory management system was extended to allow virtual address regions to be mapped as "foreign". When an attempt is made to resolve the machine address of a page mapped in a foreign region, a special code path is taken, resolving the appropriate machine address in the remote VM.

With this modification, direct block access (`O_DIRECT`) and the Linux asynchronous I/O (`libaio`) interfaces may be used in extensions on remote pages. As discussed in the performance section later in this chapter, the block backend driver has been reimplemented in userspace using the block tap, and achieves almost identical performance.

3.3.2.2 *User-Level API*

As mentioned above, messages intercepted by the block tap may be passed to extensions in userspace using communication rings and memory mapping similar to inter-VM device channels. There are two components to this interface: The low-level memory-mapped interface presented to userspace, and the library interface provided by `blktaplib`.

Raw Interface

At the most basic level, the tap device simply exports a character device node to applications. This device node (`/dev/blktap`) can be *mmap()*ed into application virtual memory. The mapped virtual memory region is sparse; it includes the rings for forwarding requests and responses to respective front and backend domains, and a large region into which foreign data pages may be mapped. As discussed above, this sparse region of virtual memory is marked as containing foreign virtual memory mappings to allow direct I/O access from extensions.

In addition to the *mmap()* interface, the device provides an *ioctl()* which may be used to signal that there are outstanding messages to be read by the driver, and a *poll()* interface to receive upcall notifications of new pending messages from the driver. Allowing applications to interact with the shared memory area directly rather than providing *read()* and *write()* interfaces preserves the decoupling of data transfer from notification described earlier in this chapter. The *ioctl()* and *poll()* interfaces combine to provide an OS-to-application analogue of the event channels used between VMs, and ensure that request batching is maintained.

In addition to signaling, the *ioctl()* interface allows the mode of the request switch to be changed, and provides an interrogation interface for debugging.

Library Support

In order to facilitate the development of extensions as user-level applications, the `libblktap` library provides higher-level interfaces to block device extensions. While this library was originally intended simply to eliminate repeated code, it has evolved over the past year to become a key component of the system.

The user-level API was initially motivated by the Linux IP chains / IP tables

interfaces, in which a series of filters and queues — effectively a list of function invocations — could be applied to packets as they were processed by a host. This approach has also been used in research systems concerned with flexible but efficient packet processing in the past [Ros94, MP96, HP91].

The original library presented two "chains", one for requests and one for responses, to which function hooks could be attached. Similar to other approaches, the hook functions would be called on a per-message basis, and would return a verdict regarding the treatment of the message. Requests could be *passed*, indicating that they should be forwarded on to the next hook, or *dropped*, indicating that processing should stop. As the block protocol is symmetric, hooks issuing a drop verdict are responsible for injecting an appropriate response back to the client VM; the drop verdict simply provides a way for a hook to consume an in-flight request.

The initial implementation has evolved in several ways. Rather than having a single set of chains for all block requests, the library now provides a virtual disk abstraction to applications. As disks are registered by the Xen control tools, the library provides a callback to indicate that a new disk has been created. Request-handling function chains are now contained within this virtual disk structure, allowing applications to build per-disk handlers to provide whatever behaviours they desire.

Figure 3.13 shows an overview of the library interface to block devices. The `register_new_blkif_hook()` call is used to register a function that will handle new disks as they are instantiated by the control tools. This function initializes necessary private state, for instance by opening the image file that will back a virtual disk, and sets function pointers to handle interrogating disk metadata. Additionally, a number of request and response message hooks are added to handle requests to the disk from the client VM. The code shown in the figure is a summarized version of the code used in the userspace block backend, described in Chapter 5.

This move to a per-disk abstraction was motivated by the language-level support for device virtualization described in [WCSG04] and [WCG04][2].

[2] Andrew Whitaker kindly sent me a copy of the Denali virtual disk source, which I did not manage to incorporate, but from which I took the notion of a per-disk abstraction for extensions.

```
/* Functions returning details on the virtual block device. */
long int get_size(blkif_t *blkif);
long int get_secsize(blkif_t *blkif);
long int get_info(blkif_t *blkif);

/* Request/response handlers.  batch_done is a hint from the
   library, indicating the last message in a batch.                */
int blkback_request(blkif_t *blkif, blkif_request_t *req,
                    int batch_done);
int blkback_response(blkif_t *blkif, blkif_response_t *rsp,
                     int batch_done);

struct blkif_ops blkback_ops = {
    get_size:    get_size,
    get_secsize: get_secsize,
    get_info:    get_info,
};

int new_blkif_handler(blkif_t *blkif)
{
    /* Here we establish any private state, open image files, etc. */
    ...

    /* Register operations for quering virtual disk metadata.      */
    blkif->ops = &blkback_ops;

    /* Finally, add handlers for requests/responses for this disk. */
    blkif_register_request_hook (blkif, blkback_request);
    blkif_register_response_hook(blkif, blkback_response);

}

int main (void) {
    ...
    register_new_blkif_hook(new_blkif_handler);
    ...
}
```

Figure 3.13: Example of the `blktaplib` Interface

3.3.3 The Network Tap

The network tap adds extension support for network devices in Xen. It was developed to explore a more general notion of device extensions after the initial block tap implementation had been completed. The ability to quickly develop network extensions has been very useful, and examples of its use are presented in Chapters 4 and 6. The traffic limiting example in Chapter 4 has recently been published at HotNets [KWC+05], and is the subject of ongoing work.

3.3.3.1 *Linux Packet Forwarding and Netfilter*

Linux has an established history of being used to build PC-based routing, NAT, firewall, and related packet processing functionality for small networks. The network subsystem of the OS is very feature-rich, providing support for filtering, limited processing, and forwarding of packets at all layers of the protocol stack. When network forwarding was moved out of Xen and into a privileged VM (between the 1.0 and 2.0 releases of Xen), the in-hypervisor packet forwarding code [WHTP02] was removed in favour of the existing Linux forwarding mechanisms.

To take advantage of the Linux forwarding tools, the backend driver proxies client VM connections as virtual interfaces in the backend VM. This leaves the backend VM with one or more physical network interfaces, handled by physical device drivers, and a set of virtual interfaces connected to individual frontend VMs. Xen installations generally use one of two forwarding techniques within the backend VM: The Linux bridge tools allow forwarding of packets at the Ethernet frame level, and include support for MAC-address filtering and proxy ARP. With a bridged interface, VMs appear as additional hosts on the local link. Alternatively, the backend VM may be configured to use the Linux routing tools, acting as an IP-level gateway for the client VMs. The bridging and routing extensions have comparable performance, and both allow the use of additional network forwarding tools, such as traffic shaping. The bridging tools are perhaps slightly easier to configure, while the routing path is undoubtedly better tested "in the wild," given the large number of home routers based on Linux.

To enable arbitrary processing of in-flight packets, Linux's netfilter interface allows packets to be forwarded to userspace over a netlink socket and pro-

```
struct ipq_handle *h;

h = ipq_create_handle(0, PF_INET);
status = ipq_set_mode(h, IPQ_COPY_PACKET, BUFSIZE);
do
{
    status = ipq_read(h, buf, BUFSIZE);
    switch (ipq_message_type(buf))
    {

        case IPQM_PACKET:
            ipq_packet_message_t *m = ipq_get_packet(buf);

            /* Arbitrary processing of the packet happens here. */

            status = ipq_set_verdict(h, m->packet_id,
                                     NF_ACCEPT, 0, NULL);
            break;
    }
} while (1);
ipq_destroy_handle(h);
```

Figure 3.14: Example of the `libipq` Interface

cessed by arbitrary applications. As mentioned above, this is the approach that was initially used to prototype network tap-based extensions, as combining this user-queueing with a VMM allows packets to be processed in userspace *outside* the client OS that they relate to.

Figure 3.14 shows a simplified version of the IP queuing example code on the `libipq` manpage[3]. Error checking has been removed to ease readability. This example demonstrates the structure of a typical packet processing application, which after initializing a connection to the ipq netlink socket, sits in a loop processing packets. The call to `ipq_set_mode()` indicates how much of the packet is to be copied to user-space; the example copies the entire packet, while it is also possible to only process packet metadata. `ipq_set_verdict()` passes a verdict on a packet, either `NF_ACCEPT` or `NF_FAIL`, to the kernel. The call may optionally include a pointer to a replacement packet to send, should the processing application desire to modify the packet contents.

[3]The `libipq` library and manpages are maintained as part of the large netfilter project. Additional information is available at `http://www.netfilter.org/`

3.3.3.2 *Structure of the Network Tap*

The network tap shares the same structure as the generalized tap interface described earlier, but uses the ring-pair-based network device channels. The tap has been incorporated as a component of the network backend driver that allows requests to be redirected through user-space. The tap and backend share a common VM, as shown in the third example in Figure 3.9.

Packets retain their batching as they are shuttled from kernel to user-space. For this reason, the tap achieves considerably higher performance than the Linux user-queuing code, which context switches at a per-packet granularity and copies packets instead of mapping them. The per-packet interface provided by `libipq` can be preserved by providing library support above the net tap, allowing `libipq`-based extensions to run.

Additionally, the grant *transfer* mechanism makes mapping data pages to user space considerably more straight forward within the tap. As the transfer of pages actually results in the tap domain *owning* the data pages, they may be mapped to user space using the existing OS mapping operations and do not require the use of OS modifications for foreign memory access as described in regard to the block tap.

3.3.4 Performance

I now consider the performance costs of the soft device architecture on real device workloads. The degrees of isolation provided by the soft device framework allows a gradient of separation for extensions both in terms of increasing safety (through isolating extension failure) and in terms of decoupling extension source from the code bases that they interact with. In the weakest form, an extension may be placed in-kernel, in the backend driver; this is effectively the same as traditional ad hoc device extensions which modify arbitrary kernel code, requiring an understanding of the kernel, and resulting in extensions that are tightly coupled with the kernel. As separation increases, extensions built on the soft device architecture may be promoted to user space and/or moved to an isolated virtual machine. These garner the benefit of additional isolation, but result in performance overhead due to the additional protection domain crossings involved.

While the benefits of isolation for safety and separation of code bases are

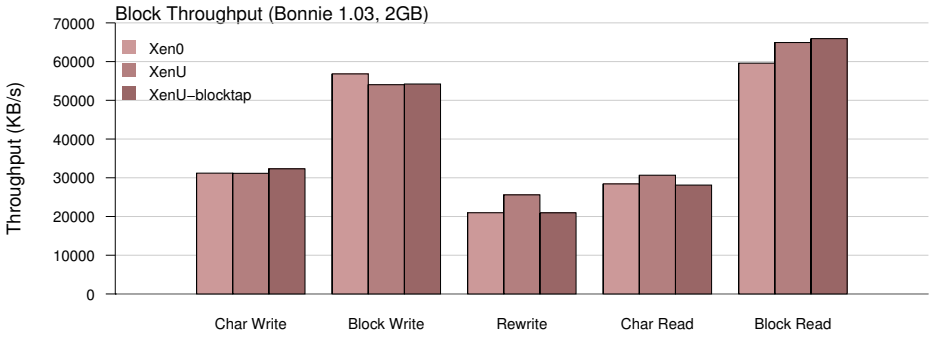


Figure 3.15: Block Throughput

rather difficult to measure empirically, the study of efficient cross domain communications has been well explored in existing systems literature, primarily with respect to microkernels [BALL90,Lie93], but also more recently with respect to Xen itself [CG05, MST+05]. One of the interesting results that we have realized in the development of Xen is that these context switch overheads can be amortized over batches of requests. As discussed in section 3.2.7 the worst workload for soft devices, and for split drivers in general, is that of small message, synchronous ping-pong style communications, where latency rather than throughput is critical. Fortunately, this class of applications is small, the only example that has been raised from the Xen community to date has been a class of MPI-based applications for scientific computing.

In [HWF+05] we argued that a key benefit of the VMM structure is that it is a practical realization of the isolation goals touted by microkernels. As argued above, building extensions at the application level in OS-granularity containers provides developers freedom of development environment and portability of extension code. The question that this section aims to evaluate is whether or not this approach to extensions is practical in real systems. To address this, I examine the overhead of soft devices on macro-benchmarks for both disk and network.

3.3.4.1 *Block Device Overhead*

Block devices are sufficiently slow relative to processor speed that achieving low overhead for extensions is relatively easy. Figure 3.15 shows bonnie

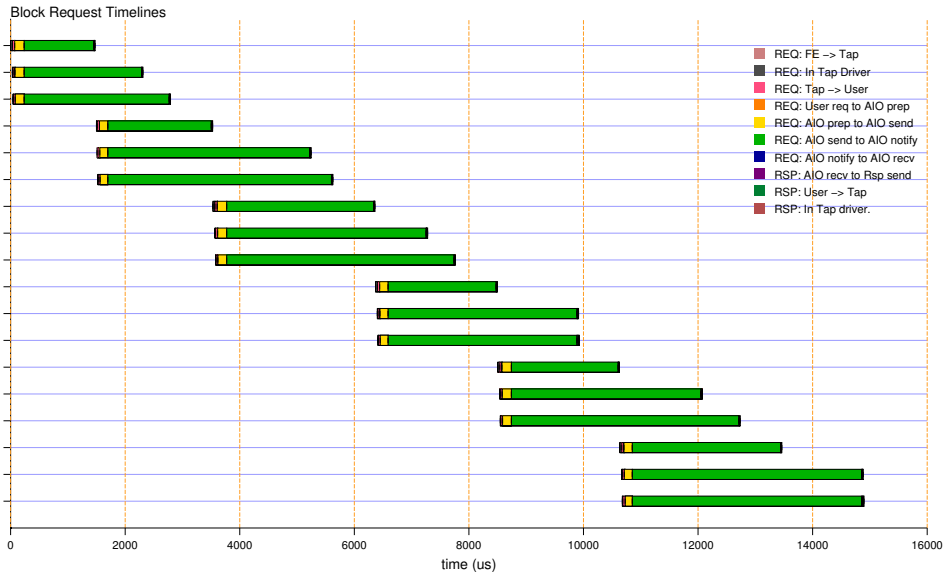


Figure 3.16: Block Request Timelines

results for the block tap. The results compare the block tap to the block backend, each serving a client VM on a second CPU. The baseline result is that of Linux running in domain 0, indicating the direct throughput to disk without using the split driver. In general, the tap is capable of saturating the block device in all configurations.

As an aside, these results show an interesting aspect of performance measurement for devices that the group has seen throughout the work on Xen: Device interactions above a VMM are sufficiently complex as to be sensitive to small changes in the environment. In the three cases below, the block tap achieves slightly better performance for small writes and block reads, while the backend driver outperforms considerably on rewrite. These results were repeatable across sets of three tests each on the same configuration, and likely represent timing and batching artifacts. Similar behaviour is shown below with regard quantizations on network receive and these device sensitivities have been discussed by other researchers in [MST+05].

Figure 3.16 presents a more detailed view of how in-flight requests are composed from a performance perspective. The diagram plots profile time-lines of a set of requests taken from a trace of sequential reads issued by the frontend. In this test, requests are passed through the block tap to a user-level

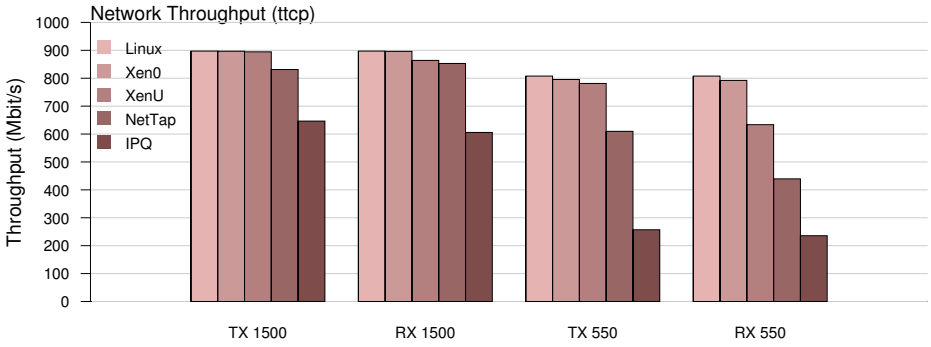


Figure 3.17: Network Throughput

implementation of the backend driver. Requests are then issued to the kernel using the Linux AIO interfaces. As this is a sequential read, each of the horizontal bars in the plot reflects a request to read 44K (11 pages) of data. This diagram illustrates how the overheads imposed by the extension interfaces are dwarfed by the time required to satisfy block I/O requests, shown as the large light-green region in the middle of each bar.

3.3.4.2 *Net Device Overhead*

The interrupt rate involved in handling modern network interfaces presents a considerably more formidable challenge for device extensions than do disks. Figure 3.17 shows throughput achieved for a 2GB network transfer using `ttcp`. As with the block throughput measurements above, all results represent the mean of three consecutive trials. The graph shows transmit and receive throughput at packet sizes of both 1500 and 550 bytes. An MTU of 1500 is the default for Ethernet, while 552 is commonly used by dial-up PPP clients and puts considerably higher stress on the I/O system due to the packet rates generated (190,000 packets per second at 800Mb/s). These values are taken from a larger set of results which characterize throughput for MTUs ranging from 1500 down to 300 and sampling at 50-byte intervals.

The first three bars in each cluster of the graph represent configurations discussed in the performance section of the presentation of split drivers. From left to right they show a normal Linux Installation (Linux), Linux running in Xen with direct hardware access (Xen0), and Linux running in a frontend VM using split drivers. As discussed earlier in this chapter, split

Configuration	Ping RTT (Overhead)
Linux on Xen – No split drivers	155μs
Split driver	206μs *(25%)*
Split Driver + Net tap	220μs *(30%)*

Table 3.2: Network Latency Overhead

drivers track the performance of native device access, but incur an overhead on packet processing of small MTUs.

The two remaining bars show the throughput achieved using the network tap (NetTap), and the Linux IPQ interface. In both cases, packets are mapped to userspace, and immediately returned to kernel by a no-op extension. While both of these incur overhead, the network tap is clearly superior. For MTUs between 1500 and 550, the IPQ throughput degrades linearly; the interface involves no batching and incurs a context switch on every packet. The network tap has much better performance: It tracks the native Linux results with an approximately seven percent overhead for MTUs down to 750 bytes, and then begins to decay as the CPU becomes saturated.

As mentioned above, the worst case workload for split drivers and tap-based extensions are those requiring low-latency synchronous traffic. This form of workload is effectively non-existent in both enterprise and desktop environments where applications are generally focused on throughput and present considerable opportunity to amortise switching overheads through batching. In order to provide some insight into the overheads on single-packet performance, Table 3.2 presents average ping times and the associated overheads using the network in three configurations. As shown in the table, the introduction of split drivers imposes a 25% overhead on ping latency on a local gigabit network. The addition of the tap, resulting in the forwarding of requests through user-space adds an additional 5% to this—context switches to user cost considerably less than world switches across VMs. These results aim to punctuate the fact that in the case of low-latency synchronous workloads—likely composed of only a very small number of scientific computing applications—the latency and CPU overheads of the techniques presented in this thesis are likely to be prohibitively expensive.

The network results demonstrate that the tap is capable of sustaining high throughput on a network saturation benchmark, and that it far outperforms

the network extension interface currently offered by Linux. While there is a higher overhead concern than exists with extending block devices, these results are reasonable to deploy extensions in production systems – especially where the network is not saturated. Moreover, network extensions should benefit from both faster processors and parallelism: moving the extension code to a separate CPU should result in a reduction in overhead, based on the experience of split drivers. The current trends in hardware are clearly towards an increase in the number of processing elements available on a host, and so this approach seems quite attractive as future work.

This concludes the presentation of support for device extensions. Tap devices extend the device channel-based communications used by split drivers to allow new functionality to be added to devices below the virtual machine. This support for the development of *soft devices* provides the following benefits:

1. **Extensions are resistant to tampering.** As device extensions execute outside of the VM that uses the device, they are considerably more secure against exploits to that OS. This is useful in adding features that enforce behaviour, such as the network traffic limiting that is presented in Chapter 4, and in building "trusted" extensions to hardware.

2. **Extensions remain co-located with the client VM.** While extensions run *outside* the virtual machine accessing the device, they are still on the same physical host. This is preferable to implementing extensions on a separate host such as a network gateway or remote storage server, as extensions are directly involved in the device interface. This provides a more immediate mechanism to provide feedback, and avoids the need to queue outstanding requests.

3. **Extension source is decoupled from front- and backend OS code.** This source isolation has two benefits. First, extensions are portable across any OS for which a device frontend is available, unlike proprietary OS-specific extension mechanisms or, worse, direct modifications to the OS kernel. Secondly, extension developers are freed from having to track changes to kernel interfaces over time.

4. **Extension execution is strongly isolated.** Finally, as extensions are written as user-space applications, and optionally run in their own vir-

tual machine, their failure is isolated from affecting the rest of the system. Extensions may easily be shut down and restarted while serving requests from running VMs.

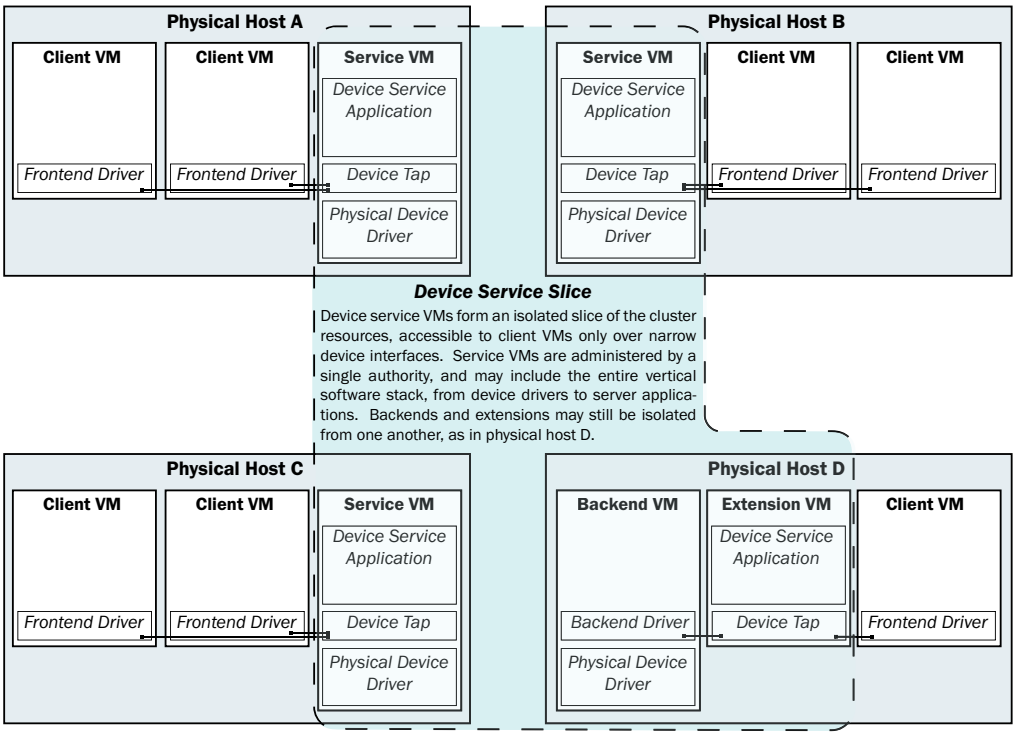
The next section explains how a set of local device extensions may be aggregated across hosts in a cluster environment to manage a class of devices, such as storage, as a service.

3.4 Device Services

In addition to allowing the extension of local devices, this thesis argues that the combination of isolating extension code in virtual machines and interfacing with client VMs on simple, narrow, virtual device interfaces provides an ideal environment for the construction of *device services*, particularly in VM hosting environments involving large numbers of physical machines.

This approach to building services behind device interfaces is beneficial from several perspectives. In the short term, many large-scale ($\sim$10K physical machines) hosting facilities are in the process of deploying virtualization. This deployment means effectively increasing the number of active operating system instances by a factor of between ten and one hundred. While virtualization allows them to increase the number of isolated servers that they are capable of hosting, it also carries the consequence of dramatically increased load on storage and network facilities. Examples such as the parallax storage service (Chapter 5) and aggregate rate limiting (Chapter 4) address immediate needs that have resulted from this sudden increase in scale.

From a more general perspective, structuring services in a data center environment as an aggregation of service VMs that interact with client VMs over device interfaces allows the isolation afforded by virtualization to be carried beyond the local host and into a facility-wide environment. After detailing the general structure of these services, this section will discuss how soft devices may be composed to form strongly isolated cluster services.



A device service is composed of a set of VMs, each serving device interfaces of client VMs on the same physical host. Device services provide performance, security, failure, and administrative isolation for a specific class of device (e.g. storage) within a cluster environment.

Figure 3.18: Structure of a Device Service

3.4.1 A Device Service Architecture

Figure 3.18 shows the structure of a generalized device service. A set of four physical hosts is shown, where a network-wide service is hosted across the entire cluster, composed of individual service VMs on each physical machine. On a given physical host, a service VM may include three main components: physical device drivers with direct access to local devices being managed as part of the service; a service application, running as a device extension in user-space of the service VM; and a client-VM-facing interface, provided by a device tap or backend driver.

Where desired, individual service VMs may be further decomposed, as mentioned earlier in this chapter, to isolate extensions into a separate VM. This is shown in the case of physical host D in Figure 3.18, where the service VM has been decomposed into a backend VM, which hosts the physical device and associated driver, and an extension VM, where the user-level ex-

tension server is run. This decomposition may be desired to further isolate failure, or to allow extensions to be developed in a completely different OS environment than the one that the physical device driver is available for.

This method of structuring device services achieves four kinds of service isolation in a VM cluster environment:

1. **Performance Isolation.** Overload of a service slice will not have cascading effects onto unrelated VMs in the cluster.
2. **Security Isolation.** Services present only device interfaces, limiting exposure to security exploitation.
3. **Failure Isolation.** Crashes and related failures in service VMs do not compromise the stability of non-dependent VMs.
4. **Administrative Isolation.** A device service represents a single cluster-wide administrative domain.

Beyond performance isolation, the remaining three categories represent the so-called "*-ilities*" of systems research; they are clearly desirable, but are difficult to validate empirically. The remainder of this section discusses each of the four in more detail and provides initial evidence to support the associated claims. In the remaining chapters of this thesis, I construct two cluster scale device services and claim, anecdotally, that they exhibit these properties.

3.4.2 Performance Isolation

One of the fundamental benefits of virtualization touted throughout the recent resurgence of research into the area has been the ability to provide strong performance isolation between VMs that are co-located on the same physical host. This isolation is of clear benefit in placing service VMs on the same physical host as clients as it is desirable to be able to ensure that the distributed device service will not detrimentally affect the clients (who are typically the paying customers) in a hosting environment.

The use of VMMs, Xen in particular, to achieve performance isolation was illustrated in our early work on the project. Specifically, [BDF+03] demonstrated system-wide performance isolation using industry standard bench-

marks such as SPECWeb.

Interestingly, while a VMM is demonstrably capable of very effectively isolating coarse-granularity workloads within a system, the accounting of some forms of device-level access remains a very difficult problem. Providing fair, and accounted access to disk in particular is a difficult problem that remains open for examination in the future.

3.4.3 Security Isolation Through Narrow Interfaces

A key aspect of this isolation is the narrow device interfaces provided to the client VMs. This is the only visible aspect of a service, and so can be more easily verified for security. A distributed service is composed of VMs that may be placed on an isolated subnet, and kept separate from any form of VM interaction. The Xen network interface already provides facilities to prevent any form of spoofing from client VMs, "stamping" outgoing packets with the appropriate MAC and IP addresses [BDF+03].

The only VM-facing interface offered by the device service is that of the device channel presented to the frontend driver in each client VM. The entire environment of device services in a cluster — client access over narrow, host-local interfaces and the ability to clearly isolate the service network from clients — allows us to avoid many of the problems faced by traditional distributed systems.

3.4.4 Failure Isolation and Fate Sharing

Just as virtualization on the single host allows us to isolate two co-located servers from all but failures in physical equipment and the VMM itself, this approach to building cluster services allows a horizontal isolation of a distributed service throughout the cluster environment.

In this manner, virtualization allows services to be composed using an isolated "slice" of the cluster, composed of a set of VMs on separate physical hosts, each serving local client VMs over local device channel connections. A novel aspect of this approach relative to other distributed or peer-to-peer service structures is that each server VM shares fate with its clients for the majority of failure conditions. For example, while it is still possible for net-

work disruptions to interfere with the distributed service cluster-wide, we have high degrees of assurance that local device channels will remain serviceable.

The use of VMMs to isolate failing software extensions, specifically device drivers, was presented in [FHN+04]. In that paper we demonstrated the ability to isolate device driver crashes, and restart driver VMs with minimal disruption to clients.

3.4.5 Administrative Isolation

In addition to the isolation of individual services from other VMs within a cluster, this approach also allows a high degree of administrative isolation, especially where service VMs actually manage physical components of each host. For example, a storage service may actually control the local disks in each physical machine within a cluster, allowing a single administrator to specialize in storage administration, having remit over all aspects of storage throughout a hosting facility. This includes the ability to administer the entire vertical stack of software pertaining to a given service, including low-level tasks such as the upgrade of device drivers, across the hosting facility. This approach is similar to that of virtual appliances [SBC+03], but applied to cluster wide services, and to lower-level interfaces.

Device services are an attractive approach to managing I/O resources in large computer installations because they allow the isolation achieved by VMMs to be applied to the management of devices. An example of this approach is Parallax, a device service for managing storage resources which aggregates local disks and network-attached storage into a single centrally-managed facility. Parallax has been initially described in a paper at HotOS [WRF+05], and is the subject of ongoing research in the lab. The system is presented in detail in Chapter 5.

3.5 Summary

This chapter has presented the core approaches taken to virtualizing and extending devices by this thesis. In a VMM environment, *split drivers* are used to isolate driver VMs, which run unmodified drivers and export them to client VMs over idealized interfaces. A *device tap* may be interposed on this interface and used to host device extensions, which may be written in userspace using high-level languages and development tools. These extensions, or *soft devices*, may then be aggregated within a cluster environment to compose *device services*, which allow the isolation provided by VMMs to be applied to the management of I/O devices across a large compute facility.

Table 3.3 summarizes the material presented in this chapter by restating the original problems as described in the introduction and briefly revisiting the key aspects of each solution. The remainder of this thesis attempts to validate these techniques by describing extensions and device services that have been constructed above these mechanisms. Chapter 4 presents a network soft device that aims to prevent denial-of-service attacks being hosted from within VMs. Chapter 5 describes Parallax, a device service for the management of storage in cluster environments. Chapter 6 describes the use of disk and network extensions to support the addition of a new architectural feature—tainted data tracking—to a virtual machine.

Device Support in a VMM (Split Drivers)	
Problem	*How should device access within a VMM be structured as to allow guest OSes to share hardware resources while balancing issues of performance, dependability, security, and software maintenance effort?*
Key aspects of split drivers as a solution:	
Multiplexing	Idealized interfaces allow new OSes to be rapidly modified to use virtualized devices. Split drivers allow multiple VMs to safely share access to a common physical device.
Source Independence	Isolating drivers in OS-granularity units and directly exporting hardware allows existing physical drivers to be used. The VMM does not need to track driver-OS interfaces, and may use drivers from any OS.
Isolation	Driver code executes in an isolated virtual machine. System is protected from most forms of driver crashes. Driver VMs may be rebooted for recovery.
Device-level extensions (Soft Devices)	
Problem	*How can device-level system interfaces be extended to allow the safe introduction of new features with reasonable performance, while making extension development both reasonably easy and portable across OSes?*
Key aspects of device taps as a solution:	
Tamper-resistance	Device extensions execute outside of the client VM and are protected against tampering, even if the OS is compromised.
Proximity	Extensions are directly associated with hardware interfaces. OS behaviour may be controlled for activities such as rate-limiting and resource control.
Source Independence	Extensions are portable across OSes. Extension developers need not track changes to OS-device interfaces over time.
Isolation	Extensions are written as user-space applications, and optionally run in their own virtual machine. Failure is isolated from affecting the rest of the system. As with split driver backends, extensions may easily be shut down and restarted while serving requests from running VMs.
Managing device access in VMM-based clusters (Device Services)	
Problem	*As virtualization is deployed into large cluster environments, how can devices be managed in a facility-wide manner, while catering to new capabilities, such as migration, that are afforded by VMMs?*
Key aspects of device services as a solution:	
Performance Isolation	Overload of a service slice will not have cascading effects onto unrelated VMs in the cluster.
Security Isolation	Services present only device interfaces, limiting exposure to security exploitation.
Failure Isolation	Crashes and related failures in service VMs do not compromise the stability of non-dependent VMs.
Administrative Isolation	A device service presents a single cluster-wide administrative domain.

Table 3.3: Summary of Approaches to the Management of Virtual Devices

Chapter 4

Soft Devices for Traffic Management

The virtualization of network resources has been of particular interest in recent years, as facilities such as virtual hosting providers and academic research networks strive to provide data-link (typically Ethernet) layer access to shared network resources in a fair, and isolated manner. My original involvement with network virtualization was with the network subsystem of Xen, which was described both as a Planetlab design note [WHTP02] and in the original Xen paper [BDF+03].

This chapter explores the application of device extensions to address problems relating to the management of network resources in VMM-based environments. As with other examples that are presented throughout this thesis, device extensions are particularly relevant in that they first present an opportunity to cleanly extend systems in a manner that has hitherto been difficult to achieve. Second, they address emerging problems that result from the scale, and division of administrative responsibility that are presented by large VMM-based environments. These two characteristics may be restated specifically in terms of network resources as follows:

1. **The ability to place isolated extensions at the network edge.**

   Soft devices are an especially powerful abstraction with regard to network resources in that, unlike in-OS extensions, they cannot be directly subverted under the compromise of administrative access on the client OS. Secondly, extensions may limit a VM's ability to transmit data by directly applying back-pressure: they may simply refuse to accept transmissions, forcing the burden of queueing onto the client OS

itself.

2. **The need to manage network resources in VM clusters.**

   The division of the administrative role that occurs in VM environments, first discussed in Chapter 2, leaves facilities administrators responsible for the management and accounting of resources available to VMs. This responsibility is particularly prevalent with regard to network resources, where administrators may be at least partially responsible for Internet traffic generated by individual VMs. As such, administrators desire techniques to ensure that VM traffic is appropriately accounted and well-tempered.

This chapter aims to present a general overview of the challenges of network device virtualization beyond those mentioned in Chapter 3. While the coverage in the previous chapter was concerned with the architecture of network virtualization support, the coverage here mentions additional issues that exist above the split driver implementation.

Additionally, this chapter presents an example that validates the ease and flexibility of development of device extensions based on the soft device framework. The example described here is that of a packet symmetry-based rate limiter, which prevents VMs from generating common forms of denial-of-service (DoS) traffic. The general approach of traffic symmetry has recently been presented at HotNets [KWC+05], while the treatment of the material presented in this chapter is specific to VMM-based environments. The symmetry-based limiter aims to provide an initial demonstration of the applicability of device extensions to solve real problems in VM environments.

4.1 Network Device Extensions in Xen

Before describing a more advanced network extension, it is worth noting that the existing network virtualization provided to VMs above Xen already augments the physical network interface with a variety of extensions.

4.1.1 Preventing Source Address Spoofing

The first example of this augmentation is in virtual network support to prevent the transmission of packets whose source addresses do not match the configuration of the virtual interface. This "antispoof" feature was demonstrated in the context of programmable network interfaces by previous research in the lab [PF01] which was in turn based on earlier packet-filter work such as DPF [EK96]. Additionally, similar features have recently been added in Windows XP Service Pack 2, which explicitly prevents the generation of UDP packets containing a source address that is not currently assigned to a physical interface.

As the network backend driver maps packets that are queued for transmission, it copies out a range of data from the beginning of each packet into its own private address space. The copied region includes the source addresses from the Ethernet and IP frames, which may be validated against the transmitting virtual interface. Maintaining this data as a private copy prevents tampering by the transmitter after validation and allows the backend to "stamp" packets with the appropriate MAC source address before transmission.

This facility has been in place since the original version of Xen. On almost all interfaces, scatter gather DMA is used to link the copied region of packet headers with the remaining payload data mapped from the transmitting VM.

4.1.2 Resource Isolation and Rate Control

The backend driver must provide fair use of network resources across the set of VMs which share a device. This enforcement is generally applied to outgoing traffic, as the physical host is effectively powerless to limit inbound traffic in a useful way. However, inbound traffic is limited in the case that a VM has not donated sufficient pages of memory to receive arriving packets: Where donated pages are not available to transfer received pages, the contained packets are simply dropped.

Packet transmission is managed by the backend driver iterating across the set of device channels with pending outbound traffic. The backend maintains

a list of channels awaiting transmission, and serves them in a round-robin fashion, dequeueing and transmitting a fixed number of packets from each interface at a time. VMs are prevented from enqueueing more packets than may fit on the transmit ring in a manner that is analogous to queueing packets for transmission on a physical device. The backend can easily throttle outbound traffic from a given VM by not dequeuing packets, and disabling event notifications on the device channel.

Additionally, the split network drivers in Xen have built-in support for leaky bucket-based, per-interface rate limiting. This support allows a hard limit to be placed on the number of bytes that a VM may transfer within a window of time, and is enforced within the backend driver.

Finally, as the network backend has been more tightly integrated with the existing Linux networking code, the Linux network management facilities are available to the control of traffic. The Linux bridge or routing utilities may be used to handle traffic forwarding, while IP tables and rate control extensions may be used to filter and shape traffic.

4.1.3 Migration of Network Addresses

When migrating virtual machines within a cluster, the virtual network interface must change physical location and its traffic must be redirected to reflect this move. In order to minimize the period of disconnection, the migrating VM carries its MAC address with it, and an unsolicited ARP advertisement is sent to indicate that the interface has moved. This technique allows a VM to be relocated within the context of a local subnet very efficiently and with a minimum of lost packets; it is one of the key techniques used to achieve the fast live migration of VMs [CFH+05].

Migrating VMs across subnets is also possible, but requires more careful integration with packet forwarding in the cluster. At the point of migration, associated routers will require reconfiguration to reflect the relocation of the migrating VM. It is very likely that enhanced support for cross-subnet migration will be developed in the near future.

4.1.4 Virtual Private Networks

Finally, support exists to use Ethernet-over-IP tunnelling to build virtual private networks between VMs. In this configuration, the backend constructs a VPN by maintaining a set of encrypted tunnels between participating physical hosts. The Ethernet link visible to a VM is then bridged over these tunnels, and physically disjoint VMs communicate as if they share a common subnet.

In all of these cases, extension code runs above the unmodified physical device driver, but outside of the isolated client VM. As extensions are frequently used to achieve security or resource-limiting goals, their placement outside the client VM ensures a great deal of resistance to from compromised or otherwise malicious VMs.

4.2 Operator Responsibility and Denial of Service

It is generally in the interest of Internet providers and hosting facilities to ensure that the network connectivity that they provide is not misused. At the present time this incentive is largely an economic one: providers negotiate the terms of upstream transit links, which typically charge based on a combination of link rate and traffic volume. As a result, hosting compromised machines that are used to perform DDoS may have a direct cost in terms of bandwidth charges to the provider.

Legally speaking, DDoS represents a very challenging situation to resolve as attackers are often difficult to identify, and the resources used are typically compromised hosts belonging to third parties. It has been suggested that service providers may have a legal responsibility to prevent their networks from being used to host denial of service attacks [PP03, Rad01], as they may be held liable as a negligent third party. To my knowledge, these assertions have not been tested in court as many claims appear to be quietly settled by lawyers and insurance companies [Sca01]. In the absence of clear regulatory guidelines, these articles all advise providers to adopt industrial "best practices", and to do everything they reasonably can to ensure that their networks are not misused. Section 4.3 describes an approach based on the *symmetry* of inbound and outbound packet counts that may be deployed as

a network extension in VM hosting environments to drastically mitigate the risk of DDoS attacks being sourced from compromised VMs.

4.3 Packet Symmetry Enforcement

As mentioned in Section 4.2 providers are under increasing pressure to ensure that compromised hosts on their networks are not used to mount DDoS attacks on others. This is especially true in hosting centres, where unlike home DSL-connected workstations, hosts are server-class machines with high-rate network connectivity. In such environments, a single compromised host can generate DDoS traffic equivalent to hundreds of DSL users.

Fortunately, the ability to place traffic management extensions outside the client OS, but on the same physical host provides administrators with a unique opportunity to manage traffic. For instance, ingress filtering, as advised by RFC 2827 [FS00], can be employed at the virtual network interface, where packet provenance may be absolutely determined at the resolution of a fully-specified IP address.

This placement of extensions on the same physical host is similar to OS-kernel-based traffic limiting techniques such as the virus throttling proposed in [Wil02], "distributed firewalls" [Bel99], and Microsoft's recently produced networking security enhancements released in Windows XP service pack 2 [And04, Mic04]. However, OS-based approaches may be circumvented in the case of a system compromise: if an exploit gives 'root' or 'Administrator' access, then an OS-based approach is effectively useless. This is demonstrably true in the case of Microsoft's extensions: SP2 limits a host to no more than 10 partially open TCP connections to different remote hosts, and disallows the use of raw sockets to generate UDP packets with IP source addresses that are not assigned to an existing interface. Binary patches are publicly available that allow both of these limitations to be removed by directly modifying the Windows network stack.

An additional, and unique benefit to limiting traffic at the virtual interface is that extensions are not required to drop packets. Instead, they may opt simply not to dequeue packets that a VM is attempting to transmit, forcing that VM to bear the burden of queueing, while avoiding drops which may

be misinterpreted as being an indication of congestion.

4.3.1 Packet Symmetry

This work focuses on the observation that well-behaved connections exhibit a balance between transmitted and received *packets*. For example, although most TCP connections exhibit unidirectional data transfer, acknowledgements still flow back from the receiver to the transmitter at a rate of approximately one ACK for every two segments received; even in cases of loss we can expect this ratio to remain relatively small.

One can argue that symmetry also holds for the vast majority of UDP flows; SunRPC protocols clearly possess symmetry, and even streaming media protocols such as RTP exhibit packet-rate symmetries of better than 6:1. It seems reasonable to argue that *any* correct Internet protocol must exhibit the flow of packets in both directions if it is to be responsive to conditions in the network or at the peer host.

In light of these observations, many forms of malicious traffic may be constrained by rigidly enforcing a threshold of outbound packet symmetry. To quantify this symmetry, the following metric compares the number of transmitted packets (tx) and received packets (rx):

$$\mathcal{S} = \log_e \left(\frac{tx + 1}{rx + 1} \right)$$

This metric for symmetry produces negative values when rx outweighs tx, positive values when tx outweighs rx, and zero in the case of perfectly balanced traffic. The absolute value of $\mathcal{S}$ measures the magnitude of the asymmetry.

This model treats the transmission of reply or acknowledgement traffic as an *implicit signal* that it is acceptable to continue to send to a specific destination. We enforce an asymmetry limit of $S \leq 2.0$, a ratio of about seven to one; this limit was established empirically through trace analysis performed by Christian Kreibich, the details of which are described in [KWC+05].

As a general architectural approach, this addition of implicit signalling based on response traffic addresses the weakness in the current Internet design

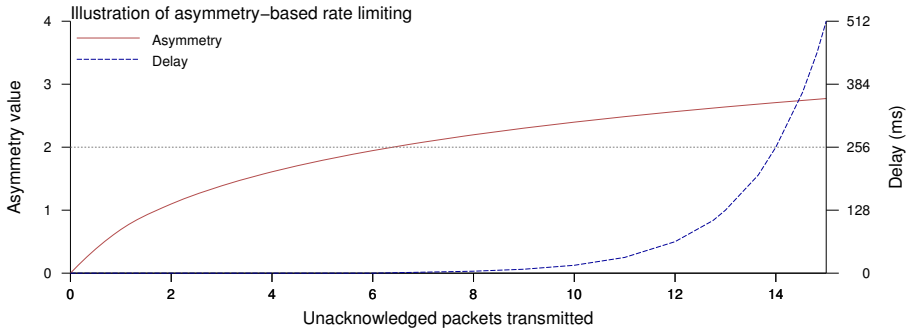


Figure 4.1: Illustration of Asymmetry-based Rate Limiting

that allows DDoS attacks to be carried out in the first place: There is no way for an end host to indicate that they do not wish to receive traffic from a given source. This inability to signal has been observed by other work, which has proposed more drastic architectural change to the network [ARW03, ALPS03, HG04]. [ARW03] for example proposes using capabilities to prevent unwanted traffic: a server must explicitly grant a client the ability to send. This explicit signalling requires modification to both ends of a connection, whereas the symmetry-based approach is incrementally deployable at the transmitting hosts.

A second major problem in DDoS prevention lies in identifying the hosts that are at the source of an attack, who frequently spoof source addresses. The difficulty in determining attack sources makes filtering attacks at the receiving end very difficult and has motivated the proposal of trace-back schemes such as [SWKA01, SPLS+02, YPS03]. The use of ingress filtering as an extension to the virtual interface provides the invariant that a VM may only generate packets from its assigned network addresses.

The asymmetry-based limiter delays, rather than drops packets at the transmitting source. The approach acts like a punitive traffic shaper: As packets exceed the threshold symmetry, it introduces a delay between transmitted packets. This delay grows in proportion to the degree of asymmetry, throttling the host's ability to generate traffic.

Figure 4.1 shows an illustration of this asymmetry-based limiter using the parameters used in the implementation described in the next section. The graph plots the increasing value of asymmetry as a host generates unac-

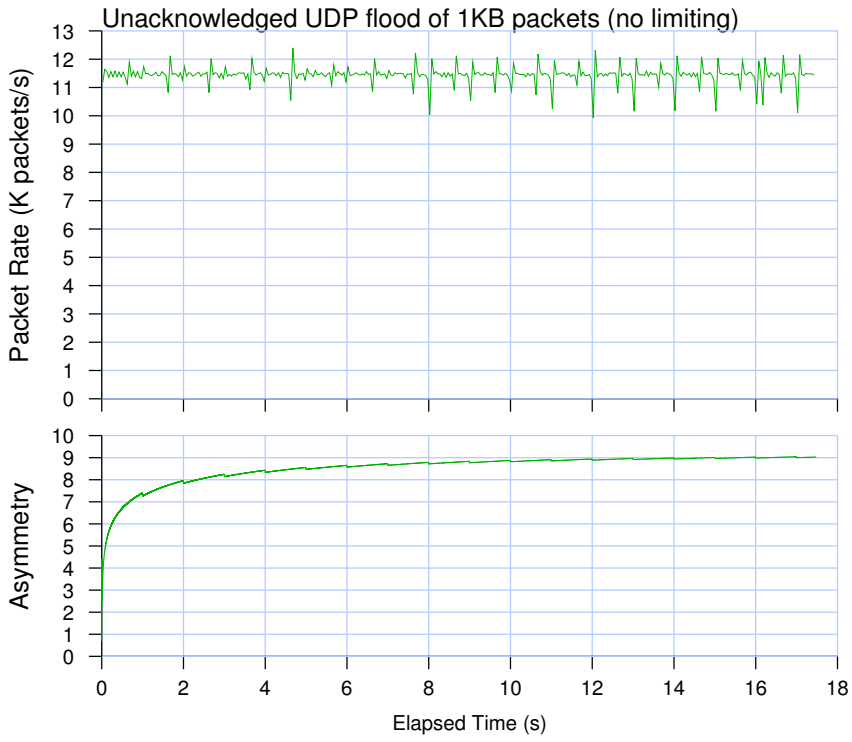


Figure 4.2: Simple DDoS Example: UDP Flood

knowledged packets. The asymmetry value crosses the threshold, marked with the horizontal line at $S = 2.0$, at which point the delay function begins to generate increasing per-packet delay penalties, for as long as the asymmetry exceeds the threshold. Asymmetry is calculated within a sliding window, so once a sufficiently large delay is generated, the host is effectively prevented from generating traffic for the length of that window.

4.3.2 Prototype Implementation

I have implemented an asymmetry-based traffic limiter as a soft device for Xen. The implementation interposes on all traffic to and from the client VM, calculating per-host-pair and per-flow symmetry values. As mentioned above, the filter is calibrated to an asymmetry value of $S = 2.0$, which is an approximately 7:1 ratio of transmitted to received packets within a one-second sliding window. This is a liberal allowance, as it gives hosts the

ability to send seven packets per second before any delay is introduced. In a real world deployment, the ratio might be reduced, and the window size could certainly be grown.

All traffic passing through the prototype with a symmetry value of 2.0 or less are dequeued and sent to the physical driver for transmission. When asymmetry exceeds 2.0, we begin to count packets as outstanding using a monotonically increasing packet counter n. Each outstanding packet is then delayed by 2^n ms before it is transmitted. This delay continues to accrue and be applied to transmit packets until the symmetry falls back below our threshold value, at which point the timer is reset. This penalty function was intended as a rough initial estimate, and will likely be reconsidered in future work on symmetry-based enforcement.

Figure 4.2 shows an example of DDoS-like traffic. The graphs plot the transmit packet rate and the resultant asymmetry of a UDP flood attack against a remote host, a simple form of bandwidth consumption DDoS. The two hosts are connected over 100Mbit links in a local LAN. The local host transmits one KB UDP packets as fast as it can, saturating the link. The symmetry value plotted in the lower graph raises logarithmically as the attack continues.

Figure 4.3 shows the effect of symmetry-enforcement on the UDP flood. As can be seen, the UDP transmission is aggressively limited at the threshold due to the lack of received packets to offset the asymmetry. Compare this to the `scp`-based file transfer shown in Figure 4.4: `scp` also saturates the link (achieving a lower packet rate because it is sending larger packets), but as it is a TCP-based protocol, stays well below the asymmetry threshold.

There are currently two implementations of the asymmetry extensions. One extends the backend network driver to enforce host-granularity asymmetry: All packets on a given virtual interface are counted together, and the host is punished for violating symmetry across the aggregate of its connections. This implementation has the benefit of being simple, and requiring very limited state. A second (original) implementation is written as an extension using the `libipq` interface, and allows per-flow symmetry to be evaluated. This requires the queuing of outstanding packets in the extension code, as they must be transferred to the extension for their five-tuple-based flow to be evaluated. A mechanism in the split network drivers to allow extensions to

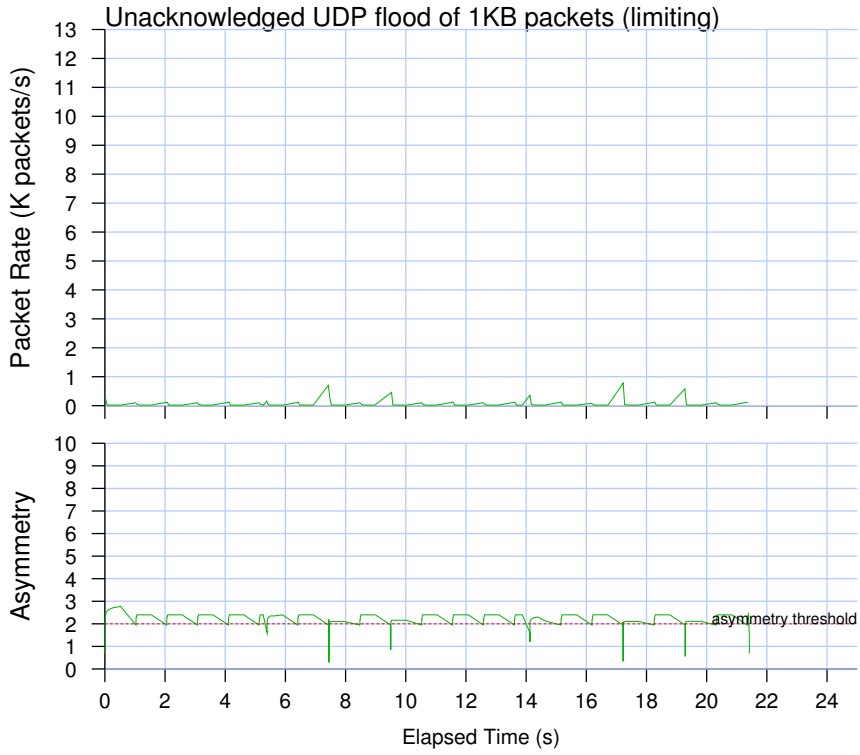


Figure 4.3: Limiting UDP Flood Based on Packet Symmetry

"peek" at pending packets, and selectively dequeue specific messages would allow VM-queuing to be preserved while enforcing flow-granularity symmetries.

While the work here describes symmetry enforcement in the context of a VM environment, my work on this project with other members of the research group, described in [KWC+05], considers symmetry in the larger context of the global Internet. I intend to continue investigating this subject in the future, as we plan to deploy a symmetry shaping device to limit external traffic on a college network.

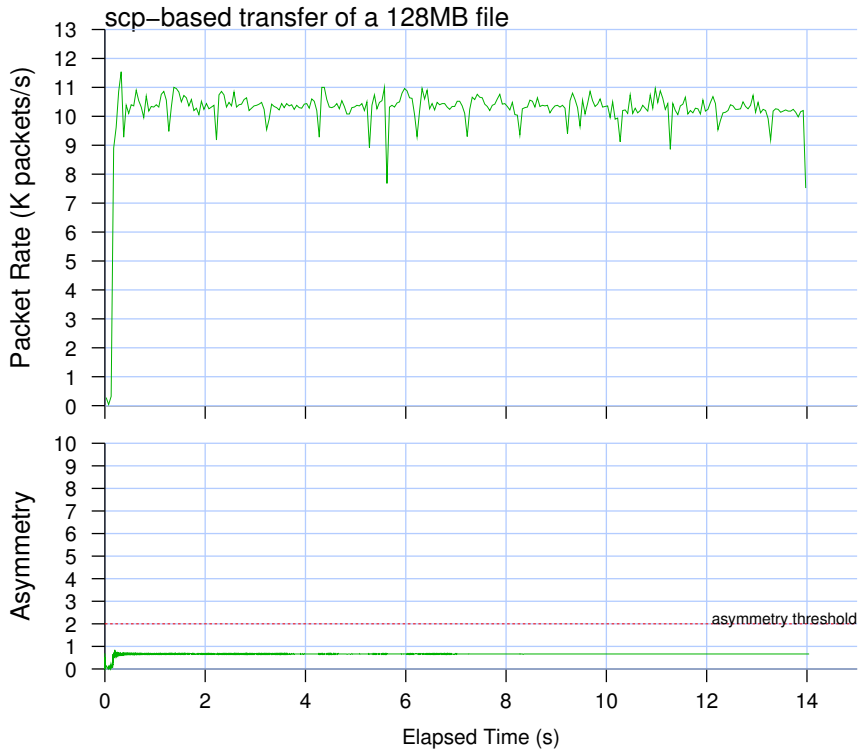


Figure 4.4: High Throughput TCP Traffic Remains Unaffected

4.4 Summary

This section has presented an overview of a set of network device modifications that are beneficial in supporting virtual network interfaces for virtual machines. These environments provide both challenges and opportunities that do not exist in existing hosting environments: as virtual machines are deployed in large cluster environments, network access must support VM migration and operators must become concerned with the increasing potential for compromised facilities to be exploited for the generation of malicious traffic. Device extensions allow the introduction of new functionalities that respond to these challenges below a VM's device interface. As such, extensions may efficiently interact with the VM, while remaining isolated from exploitation.

A future area of interest in this work is the development of a device service

to allow the accounting and rate-limiting of traffic generated by groups of VMs, as they run across a subset of physical hosts. This service would involve the development of a distributed version of a rate-control algorithm such as leaky bucket. Initial investigations in this vein have shown that while it is easy to rigidly divide a resource allocation (e.g. network throughput) across a set of distributed hosts, giving each a share, that this allocation is clearly suboptimal. A better solution would allow individual hosts to sequentially transmit at the aggregate maximum, but still reactively limit them in the case where the maximum is exceeded. The development of this thesis has involved some initial investigation into this problem, but further design and implementation is left as future work.

The symmetry-based rate-limiter is an example of a relatively simple, *practical* device extension that is beneficial in a virtual machine environment. It has been recently published at HotNets [KWC+05], and is the subject of ongoing investigation in the lab. The next chapter switches focus to explore extensions for block devices, focusing on a device service that provides storage for virtual machines.

Chapter 5

Soft Devices for Storage

As with network resources, the move to VMM-based environments represents a fundamental shift in the requirements for storage, both in terms of mechanistic issues relating to scale and bandwidth, and administrative requirements for describing and deploying storage resources. Despite offerings of "storage virtualization" products, almost all storage products that have been available until very recently have focused exclusively on the unification of a set of physical devices into a storage substrate that can be divided and exported as logical disks. Virtual machines have placed new requirements on storage in several dimensions:

- **Capacity.** Capacity is a key issue in the deployment of hardware virtualization. First, as physical servers are capable of hosting an order of magnitude increase in virtual machines, there is a concurrent requirement for system images for these virtual hosts. Second, live migration introduces a requirement that storage be more available throughout a cluster, potentially introducing a need for additional replication of data. Thirdly, if techniques for debugging [KDC05], diagnosis [WCG04], and improved availability [QTSZ05] gain traction in deployed systems, there is a requirement for very fine-grained snapshots of entire system images to be available in an online fashion.
- **Bandwidth.** As with capacity requirements, an increase in the number of OS-granularity storage clients has a direct impact on storage bandwidth requirements. This is particularly true during physical reboots and the execution of timed maintenance tasks.
- **Location Transparency.** As suggested above, live VM migration implicitly requires that storage be available in a location transparent

fashion. While network-attached storage provides a high degree of network transparency by nature, OSes have not traditionally changed physical location within the network and configuration management for this mobility must be addressed.

- **Administration.** Finally, the division of administrative roles described previously has clear implications of storage. While system administrators managing VMs may take a large amount of responsibility for the installation and management of system software, facilities administrators require tools to create and manage logical storage quickly and efficiently, to account for its use, and potentially to guard against systems which are exposed to threat due to the presence of vulnerable software.

This chapter expands on the storage requirements presented by virtual machines and details approaches which have been taken in current systems. It then introduces the Parallax storage system as an example of a device service as introduced in Chapter 3. Parallax is a storage service designed specifically for VMM-based clusters, motivated by the storage requirements mentioned above. A prototype of the system was described in a paper at HotOS [WRF+05], and this chapter presents a detailed description of a more complete implementation than presented previously.

5.1 Parallax: A Distributed Storage Service for Virtual Machines

Parallax is a device service, based on per-physical-host soft device extensions that aims to address the storage problems presented above. Parallax is based on a storage abstraction called a Cluster Virtual Disk(CVD). A CVD is a virtual storage device that is available throughout a cluster, and provides a mapping of physical blocks on an underlying storage substrate into a contiguous set of virtual blocks, which are exported directly to a virtual machine. The CVD abstraction takes advantage of the commonality between disks used by VMs to aggressively share common blocks across disks, and to easily create new disks based on well-known images.

While a prevalent issue in existing systems, concerns regarding capacity and

bandwidth are only the immediate problem in VMM-based environments. The design of Parallax aims to address the emerging research applications of virtual machines that do not simply require support for scaling capacity and bandwidth. Two applications in particular, large honey farms and VM replay, require fast and efficient snapshots of virtual disk images.

Honey farms and Application Sandboxes

VMs have attracted considerable attention from the security community as a means for intrusion detection. The Collapsar [JX04] project uses virtualization to turn a single physical host into a honey farm of virtual hosts, allowing the logging and detection of network-based attacks. More recently, the Collaborative Centre for Internet Epidemiology and Defenses (CCIED) has undertaken the construction of a VM-based honey farm containing a million virtual hosts [VMC+05], where the creation of individual VMs is triggered based on packet arrival.

In these situations, the use of VMs results in a dramatic increase in the number of system images that are required. Fortunately, the required VM images are very similar and it is consequentially possible to share the common portions. It may often, for instance, be desirable to manage numbers of disks based on a small set of template images.

One of the key aspects of the CVD primitive is a copy-on-write block-level virtualization of underlying storage. The implementation discussed in Section 4.3.2 is capable of generating 10,000 new disks from a given template in under 40 seconds.

Replay: Forensics and Debugging

A growing set of OS research efforts uses 'time-travel'—the ability to rewind and replay a VM through historical states—in order to add new functionality. In this area, virtualization has been used both for the forensic examination of historical system state, and for analyzing state changes over time to isolate the root-cause of failures.

BackTracker [KC03] is a system that uses virtualization to isolate and analyze intrusions into a virtualized OS. More recently, the same set of researchers have developed a time-travel debugger [KDC05] that allows a system to be rewound, instruction by instruction, after a crash. The Chronus system [WCG04], developed at the University of Washington, allows ad-

ministrators to apply a probe over historical versions of a machine's disk in order to isolate the point at which a malfunction began. Chronus then isolates the disk request that was responsible, in hopes that higher-level tools might be able to resolve the problem.

In addition to these examples, there are several other clear opportunities to take advantage of time-travel and the existence of past system states. I expect that cluster-wide snapshot and replay tools will be available for VM clusters over the next few years.

Preserving old state and allowing VMs to move forward and backward through time presents a very interesting set of challenges for storage. The ability to snapshot a virtual disk at runtime must be provided with virtually no overhead. Furthermore, the system must be able to store and manage vast numbers of snapshot versions. As a goal in this regard, and in light of experiences gained through VM replay efforts at Cambridge, the Parallax design assumed a goal of performing per-VM snapshots every 30 seconds. At this rate, the system must account for the creation of almost three-thousand snapshots per-CVD, per-day. As such, the snapshot mechanism has been designed to be as space- and time-efficient as possible.

5.1.1 Overall System Design

This section presents the design of Parallax, a distributed storage management service for clusters of virtual machines. Parallax includes three components: First, a distributed block store providing traditional storage virtualization, location transparency, and redundancy. Second, a middle-layer persistent cache to mitigate the performance overheads involved in storage virtualization. Finally, Parallax builds CVD support above these underlying block layers and presents access and management interfaces to virtual machines. The design is motivated specifically by the requirements and examples presented above.

The design of both Parallax and the CVD primitive is based on a set of simplifying assumptions, specific to virtual machines, that aim to provide a system that is considerably more flexible and scalable than previous approaches to storage management. This section begins with a discussion of these design decisions.

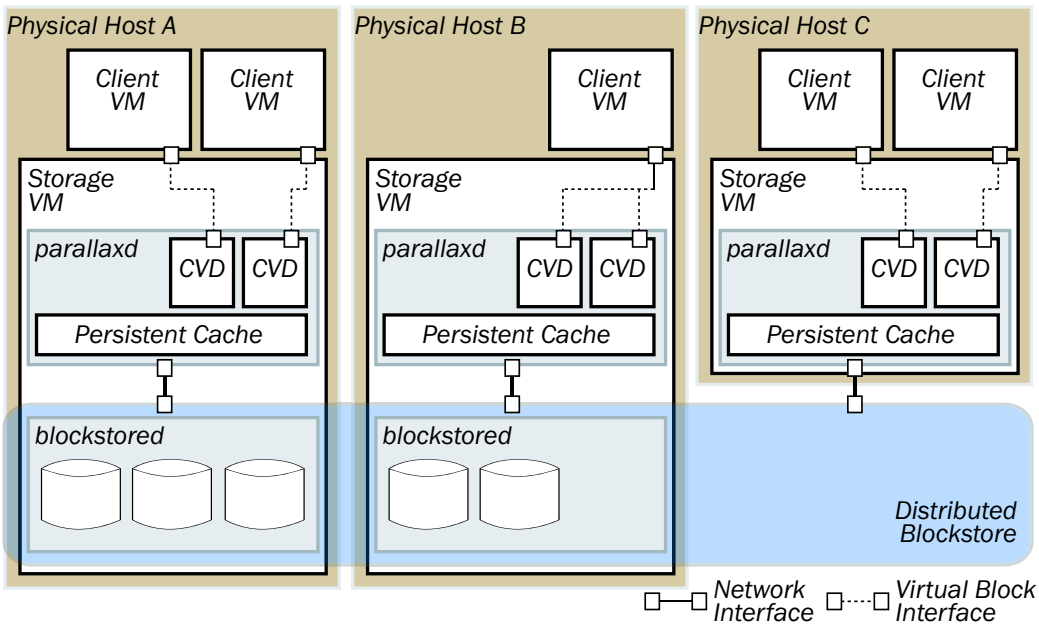


Figure 5.1: Parallax High-Level Architecture

5.1.1.1 *Simplifying Assumptions*

Virtual machine clusters present a new set of storage requirements but also have several properties that greatly simplify the design of a storage system. Parallax's design is based on two initial assumptions specific to the provision of whole system images in VMM environments.

Write Sharing Considered Harmful

Distributed storage has historically implied some degree of concurrency control. Write sharing of disk data, especially at the file system level, typically involves the introduction of some form of distributed lock manager. Lock management is a very complex service to provide in a distributed setting and is notorious for difficult failure cases and recovery mechanisms. Moreover, resolving write conflicts is a well-investigated area of systems research, and one for which no general solutions exist.

Parallax's design is based on the belief that persistent data sharing in a cluster environment should be provided by application-level services with clearly defined APIs, where concurrency and conflicts may be managed with application semantics in mind. As such, the system explicitly excludes support

for write sharing of CVD images. Instead the system maintains the invariant that each CVD has at most one writer, greatly reducing the need for concurrency control. No communication between virtual machines is required to access CVD images, only between the virtual machine using the CVD and the block store. If virtual machines wish to communicate or share data, they simply use explicit network-facing protocols rather than implicit shared-file semantics. By avoiding write sharing completely, the state shared across hosts is limited to mostly-static catalogues of storage serving hosts, which only need updating in response to high-level administrative operations.

The reduction of complexity resulting from the removal of write sharing is profound. Unlike Petal [LT96], Parallax does not maintain cluster-wide global data structures for block allocation and concurrency control. The more recent federated array of bricks (FAB) project, directly motivated by Petal, mandates the use of voting protocols for consistency [SFV+04] and requires an intricate snapshot algorithm [Ji05].

Deferred Redundancy

The virtualization of any address space involves the use of address mapping, which in turn requires extra lookup steps. A difficulty for providing such mappings, for disk especially, is that map lookups have the potential to incur additional device accesses that can greatly hinder performance. The persistent cache in Parallax provides an interposition point that gains improved performance.

Thus the system is able to cache both blocks and mapping data locally while replicated writes to the block store proceed in the background. Ongoing work with the system will involve enhancing the block-device interface to include synchronization and reordering barriers. These barriers may be used to specify when it is safe to defer write-back or to perform parallel data accesses, and when it is required that data is written back to the redundant store to enforce higher-level semantics. Through their direct exposure to this call, VMs will have the ability to selectively trade performance for availability guarantees.

5.1.1.2 *Cluster Virtual Disks*

Cluster virtual disks are the fundamental storage unit used by virtual machines in Parallax. A CVD is a single-writer extent of virtual storage, acces-

read(*vblock*) → *data*	Return the data stored at the specified virtual block address.
write(*vblock*, *data*)	Write the data provided at the specified virtual block address.
cvd_sync()	Flush all outstanding writes to redundant store.
cvd_snapshot() → *snap_id*	Take a snapshot of the current CVD image.

Table 5.1: VM Interfaces to CVDs

sible anywhere within the cluster. CVDs present virtual machines with the interface in Table 5.1.

The *read()* and *write()* interfaces reflect the minimal block device interface expected by a VM. CVDs additionally allow paravirtualized VMs to request that data be explicitly flushed to the distributed store using *cvd_sync()*. Finally the *cvd_snapshot()* operation marks the current state of the disk as a snapshot in the CVD history.

Virtual Block Mappings

The set of virtual block mappings in a CVD is stored in a radix tree that resides in the same block store as the CVD contents. It maps contiguous virtual block numbers from a single CVD to global block addresses, which identify the block and its replicas in the distributed block store. Global block addresses in the current implementation are 128 bits[1] and the size of the virtual block numbers is set at CVD creation time to the minimum size necessary to fully address that image. With four kilobyte blocks, a radix tree with a depth of three blocks is sufficient to map a 24-bit address space, supporting CVDs up to 68 gigabytes in size.

Though the radix tree is stored in the distributed block store, it is also partly duplicated in the local persistent cache and obeys the no-write-sharing constraint. The root block of the radix tree and other blocks mapping active regions of the CVD are cached in memory as well, so no disk or network activity is required to resolve most global block address lookups. In particular, virtually sequential data blocks share radix tree pages, resulting in good performance for sequential lookups.

[1]One bit is reserved by the current radix tree implementation to distinguish between read-only blocks inherited from an earlier snapshot or template and writable blocks created as part of the currently active CVD.

create(*name, (snapshot)*) → *CVD_id*	Create a new CVD, optionally based on an existing snapshot. The provided name is for administrative convenience, whereas the returned CVD identifier is globally unique.
snapshot(*CVD_id*) → *snap_id*	Force a snapshot of the specified CVD. This is an administrative interface to the CVD snapshot facility described in Table 5.1.
list() → *CVD_list*	Return a list of CVDs in the system.
snap_list(*CVD_id*) → *snap_list*	Return the log of snapshots associated with the specified CVD.
snap_label(*snap_id, name*)	Label a snapshot with a human-readable name and mark it `forkable`.
snap_collapse(*parent_snap_id, child_snap_id*)	Where the parent and child are co-linear entries in the snapshot log, this deletes all snapshots beginning with the parent and exclusive of the child..
delete(*CVD_id*)	Delete the specified CVD. All snapshots are removed backward in time to the last forkable snapshot.
tree() → *(tree view of CVDs)*	Produce a diagram of the current system-wide CVD tree (see Figure 5.3 for an example.)

Table 5.2: Administrative Interfaces to CVDs

Snapshots

The root node of a radix tree uniquely identifies the CVD to which it applies. This property facilitates support for lightweight snapshots, which require only a single block duplication. The snapshot becomes a read-only template from which the active CVD diverges over time using copy-on-write. Careful construction of metadata allows this to proceed in a space- and time-efficient manner.

A flag on each link in the radix tree indicates whether or not the target block (which may be either data or a reference to a successive level of the tree) is directly writable. All blocks created before the most recent snapshot are immutable, while those created in the context of the active CVD can be modified in-place. The links in the old root block are not changed, so writable links provide a convenient way to identify all blocks that were created during a specific CVD generation. To take a snapshot of a CVD requires making a copy of its radix tree root block and marking all of its

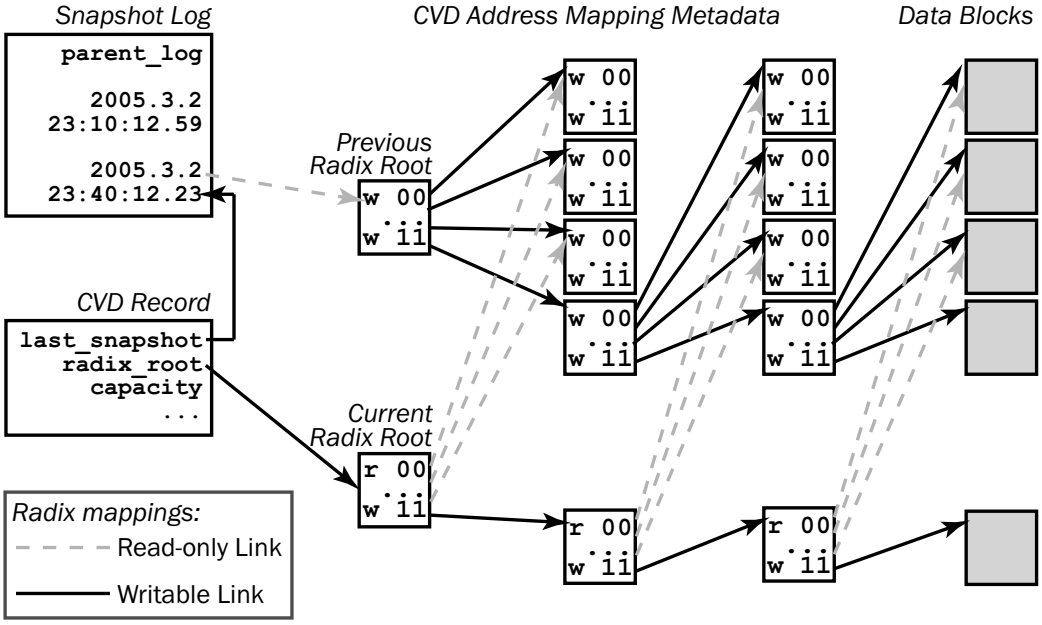


Figure 5.2: CVD Snapshot and Copy-on-Write

links as read-only. Every link on the path from the root of the radix tree to the data block must be writable for the CVD to modify a block directly, so this forces the CVD to copy blocks when it makes changes, including those for the relevant path down the radix tree.

Figure 5.2 illustrates how the radix tree block mapping structure provides snapshots and copy-on-write block access for CVDs. The figure shows a simplified radix tree mapping six-bit block addresses with two address bits per radix page. The example shows a CVD that has had a snapshot taken, and successively written to a block of data at virtual block address `0x111111`.

The snapshot operation copies the radix tree root block and redirects the CVD record to point to the new root. All of the links from the new root are made read-only, as indicated by the "`r`" flags and the dashed grey arrows in the diagram. The address of the old root is appended, along with the current time-stamp, to a snapshot log using a read-only link. The same writable-link semantics that govern links within a radix tree apply to the CVD record and snapshot logs and so a CVD mounted using a read-only link is immutable and changes to it must be made in a copy-on-write fashion.

Entries in snapshot logs may be mounted read-only for the purposes of VM replay.

Copying a radix tree block always involves marking all links from that block as read-only. A snapshot is completed using one such block copy operation, following which the VM continues to run using the new radix tree root. At this point, data writes may not be applied in-place as there is not a direct path of writable links from the root to any data block. The write operation shown in the figure copies every radix tree block along the path from the root to the data (two blocks in this example) and the newly-copied branch of the radix tree is linked to a freshly allocated data block. All links to newly allocated (or copied) blocks are writable links, allowing successive writes to the same or nearby data blocks to proceed with in-place modification of the radix tree. The active CVD that results is a copy-on-write version of the previous snapshot.

5.1.1.3 *Administrative Interfaces*

The mapping and snapshot mechanisms act to maintain the invariant that a CVD is a chain of read-only radix tree snapshots terminated by a single writable radix tree. The distinction between a snapshot and a new CVD forked from a template image is purely administrative; the same mechanism is used in each case. My prototype installation maintains a set of well-known "pristine" OS installations (FC2, FC3, Debian Sarge, and NetBSD) and generates new CVDs based on these.

The administrative CVD interface is shown in Table 5.2. The CVD interfaces are made available through a set of command-line tools available in administrative VMs. The implementation also provides some simple utility tools, used to populate CVDs with existing images. To create a new CVD, the *create()* interface is called and a name is provided to identify the CVD for convenience. The create call returns a unique CVD identifier, which may be used by a VM to mount the image. Newly created CVDs that are not based on an existing snapshot have empty radix root blocks. They occupy no additional space on the system, similar to a sparse file in most file systems.

Once a functional image has been created and properly configured, the *snap_label* interface may be used to mark it as well-known. This is partly an

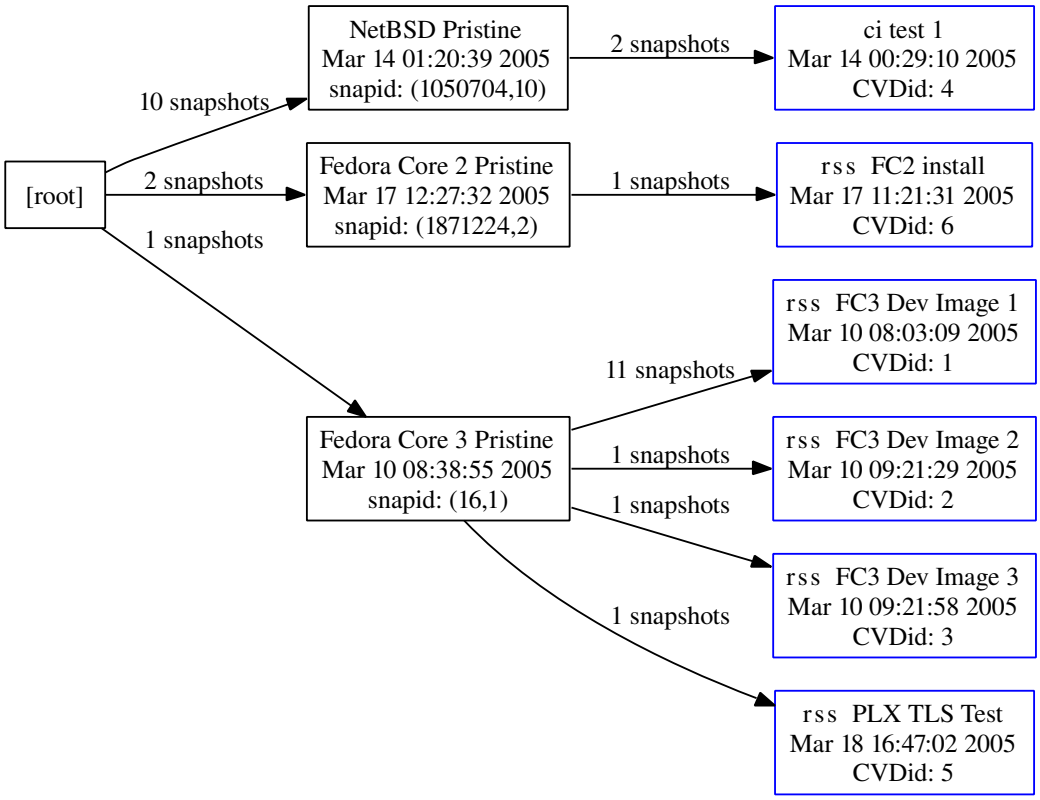


Figure 5.3: CVD Tree View—Visualizing the Snapshot Log

administrative convenience, as CVD creation is based on system-wide snapshot IDs. The list of well-known snapshots is useful for quickly locating and creating a new CVD.

Figure 5.3 shows sample output from the *tree()* interface[2]. The figure is a simplified example showing 6 CVDs, each based on one of three labeled snapshots. The tree tool, which uses AT&T's *graphviz*[3] to represent the CVD hierarchy, collapses snapshots which do not represent forks and provides an easy-to-read representation of the current storage system state.

[2]The cautious reader will notice that some timestamps in the CVD nodes precede the snapshot nodes on which they are based. This is due to the fact that the leaf node time-stamp represents creation time, while the template time-stamp is snapshot time. The earlier leafs represent the initial CVDs used to create the template snapshot.

[3]`http://www.graphviz.org/`

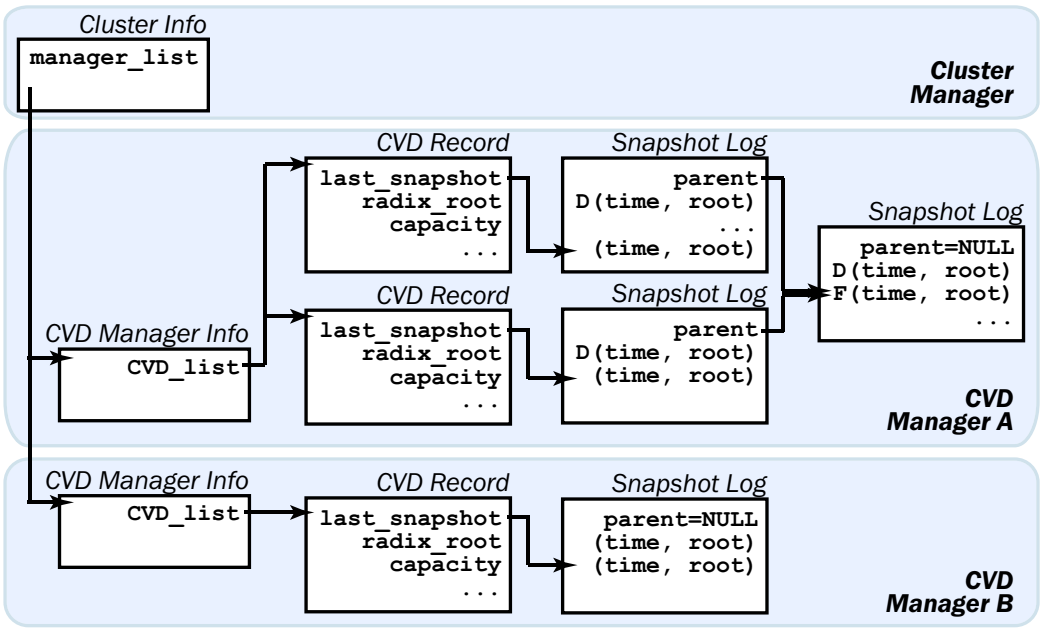


Figure 5.4: Administrative Data Structures

5.1.1.4 *Administrative Data Structures*

The block metadata structures within Parallax have been designed to allow a set of physically distributed storage servers and virtual machines to share access to a common global block address space without necessitating a complicated distributed lock manager. While read and write operations made by individual VMs are lock-free, several administrative-timescale concurrency control checks are instituted, primarily for system safety. None of these mechanisms functions at a granularity that introduces either performance overhead or significant complexity.

Figure 5.4 shows an overview of the important block metadata structures. This figure is a higher-level view of the per-CVD mapping metadata illustrated in Figure 5.2. Block metadata is divided into a set of concurrency domains, where each domain has a single writer. The top of the figure shows a single, cluster-wide data structure (*Cluster Info*) which presents a list of "cluster managers". Each cluster manager is responsible for the metadata for a set of CVDs and their associated snapshot histories. A cluster manager exports the set of administrative interfaces described in Table 5.2; since it is the single writer for this entire set of metadata, it can be guaranteed safe

access.

When a VM mounts a CVD, the cluster manager delegates write-ownership of that CVD's block metadata to it. The VM now becomes the single point of access for administrative interfaces to the CVD for the duration of the mount. In the figure, none of the CVDs are active, and so they all exist within the concurrency domains of their associated CVD Managers.

The only piece of metadata that is potentially shared between writers for common administrative operations is the snapshot log. Recall that the log is a tree of timestamped pointers to individual radix roots, each entry representing a complete immutable file system snapshot. (An example snapshot log is shown in Figure 5.3.) As operations are provided both to create a new CVD based on an existing snapshot (resulting in a fork in the log), and to delete groups of snapshots, the system must be careful to avoid conflicts.

This problem is addressed by adding two bits to each entry in the log, the `forkable` and the `deleted` bits. As its name suggests, the `deleted` bit indicates that a snapshot is no longer present in the system. The `forkable` bit indicates exactly the opposite: that a specific snapshot is guaranteed to be available. The `forkable` bit is set using the *snap_label()* interface, and allows administrators to safely create new CVDs based on snapshots to which they do not presently have write access.

Administrative operations that modify the snapshot logs may only be applied by the single writer of the CVD owning that portion of the log. Snapshot log blocks are owned by the CVD that creates them; for example in Figure 5.3, the snapshot log belonging to "FC3 Dev Image 1" extends back to the root, including the template snapshot, "Fedora Core 3 Pristine". "FC3 Dev Image 2" has been created based on this image, and so will only own the portion of the log back to, but exclusive of, this shared forkable entry. Note that the depiction of the snapshot log as a tree is slightly misleading: there is no distinguished root node, nor are there forward pointers. Instead, the tree is implicitly traversed by tracing the snapshot logs backward from each CVD's active generation.

5.1.1.5 *Deletion*

The frequent creation of snapshots has the potential to consume a considerable amount of additional disk space. The original design was inspired

by the arguments presented by Venti [QD02]: that storage was sufficiently inexpensive and plentiful that deletion is unnecessary. However, after some practical experience it became obvious that a delete mechanism would be beneficial. In a production setting the claims made by Venti are realistic. However, administrative policy may dictate that historical versions of data be removed after some period of time.

As shown in Table 5.2, two notions of deletion are provided at the interface level. In many cases, the desired action is simply to reclaim disk space by removing redundant snapshots. This approach has been discussed historically in the context of versioning file systems such as Elephant [SFH+99]. In more extreme cases, an administrator may desire to remove an entire CVD, additionally deleting all snapshots associated with it. The latter problem is very straightforward, as CVDs are single-writer leaf nodes on the snapshot log. Their deletion involves first removing associated snapshots owned by the CVD and then destroying the CVD itself.

Removing a snapshot can be described as a radix collapse operation. In the base case of two neighboring parent and child snapshots with radix root nodes S_p and S_c respectively, the operation *collapse(S_p, S_c)* will delete the parent snapshot, removing all blocks that are not also used by the child; in essence, blocks which have been *overwritten* by the child may safely be removed from the parent.

The collapse algorithm relies on the observation that a writable path from the radix root to an arbitrary block indicates that that block is *owned* by the root. The algorithm performs a simple pairwise depth-first search of the writable links in the two radix trees: links that are writable in both trees may be deleted from the parent, while links writable only in the parent are inherited by the child. All other links are left untouched.

Collapsing a range of snapshots is a simple extension of this approach. If the parent and child are co-linear entries in the snapshot log rather than immediate neighbors, deletion involves an iterative application of *collapse()* on the specified child and its immediate parent, discarding the then-deleted parent, until the specified parent is reached. As mentioned previously, entries in the snapshot log are not removed, but rather have their `deleted` bit set. Delete operations may not safely be applied across `forkable` snapshots, as there may be other CVDs referring to blocks in these snapshots.

5.1.1.6 *Distributed Block Store*

The current implementation of Parallax uses a relatively simple distributed block store. The CVD primitives and persistent caching are intended to function above any block-level storage substrate and so apply equally in systems where network attached storage is available. In light of this, the distributed block store is intended for environments where a simple storage substrate is desired over a set of commodity disks.

Storage is configured by building replica groups – collections of physical hosts which will store replicated copies of block data. Each storage VM maintains a list of replica groups available within the cluster, and is able to interact with CVDs within these groups. At the moment, a CVD must exist entirely within a replica group; a full image copy is required to move a disk to a new group. This limitation will be addressed in a future version of the block store.

Block servers within each replica group allow the allocation of virtual extents. A storage server will allocate a set of virtual extents and associate these with a CVD. All writes to the CVD are effectively block appends to the virtual extent. Virtual block addresses in the CVD are mapped, through the radix tree, to a global block address representing the block offsets into a set of virtual extents stored on servers within a replica group. The current implementation uses groups of three servers.

Given the variety of distributed block-level substrates currently available, both commercial and open-source software, the focus of Parallax has been on the higher-level management of virtual disks through the design and support of the CVD primitive. As such, the current distributed block store is complete enough to be functional but has not been a focal point within the system. Future development of Parallax will explore a more complete block store and also proper integration with existing network attached storage.

5.1.1.7 *Persistent Cache*

The final component of Parallax is the per-machine persistent cache, which interposes between CVDs and the distributed block store to provide higher performance for frequently accessed blocks. The cache is quite simple since all CVD information, except for administrative metadata such as the snapshot log, has at most one active writer. This allows blocks to be directly

modified in the local persistent cache with no need for locking or invalidation of remote copies.

The cache is of fixed size and maintains a list of each cached block, in LRU order. Read requests are satisfied locally if the block is present in the cache. If the block is not present then it is fetched into the cache and then serviced as for a cache hit.

Modifications are always first written to the cache; writing back to the replicated block store may then be deferred to trade reliability against performance. If a VM synchronously writes its modifications back to the block store then the replicated image is always up to date if the VM crashes (apart from dirty pages in the guest OS's buffer cache); however, performance will suffer because ultimately the write bandwidth is limited to that of the block store rather than the local disk. If writes are deferred for some time then modifications can be acknowledged more rapidly to the VM at the cost of seeing a somewhat time-delayed snapshot if the VM fails.

5.1.2 Implementation and Evaluation

Having presented the design of Parallax, this section presents details of the implementation of the system as a device service composed of device extensions.

5.1.2.1 *The Parallax Daemon*

The Parallax daemon is a user-level application providing CVD functionality and managing local disks. It is written using the block tap interface and currently totals about seven thousand lines of commented C source. The daemon handles block requests from locally connected VMs, provides a local persistent cache, and dispatches requests to the distributed block store.

5.1.2.2 *The Blockstore Daemon*

The blockstore daemon manages one or more physical disk volumes exported to CVDs within Parallax. Each physical host contains a block management VM, which has physical access to the exported disks and runs both of the daemons. The blockstore daemon is about fifteen hundred lines of commented C source.

5.1.2.3 *Administrative Tools*

In addition to the two daemons, a set of command-line tools are available to perform administrative tasks. These tools are small single-purpose utilities, and map directly to the set of administrative tasks described in Table 5.2.

5.1.2.4 *The Image Lifecycle*

This section considers the series of operations applied to create and manage CVDs throughout their lifetimes. The deployment of a new disk image is presented, as are the relevant performance figures as they arise. The current implementation is fully functional in terms of the CVD features described in Section 5.1.1, but has not been optimized for performance; while the current throughput figures do compare reasonably with network disk access over the open-source Cisco iSCSI initiator, I expect them to improve drastically with further effort.

Composing and Deploying

Starting with a new system image for deployment in the cluster, a new CVD is created. It is then populated using the `cvd_fill` command.

```
$ cvd_create "FC3 deploy CVD"
Created image id 532.
$ cvd_fill 532 fc3.ext3.image
filling CVD 532 with fc3.ext3.image
done (4294971392 bytes).
```

The `cvd_fill` command takes a local image file and copies it directly into the distributed block store. When it is complete, a snapshot of the new image is taken, it is labeled as well-known, and a set of new CVDs may be created based on it.

```
$ cvd_snap 532
Snapshot id is 12094639439883 0.
$ cvd_label "FC3 Pristine" 12094639439883 0
Well-known snapshot "FC3 Pristine" created.
$ cvd_dup "FC3 Pristine" "FC3 vol " 500
Created 500 CVDs based on "FC3 Pristine"
   (533 - 1032).
```

`cvd_dup` is a simple shell script that iteratively calls `cvd_create` to generate a large number of disks based on an existing snapshot. CVD creation simply

involves writing new CVD records and copying the existing radix root node: the current implementation can create ten thousand usable CVDs based on a template in under 40 seconds.

Utilization

Virtual machines may now be started based on the newly created CVDs. Current measured disk performance is shown in Figure 5.5. The figures shown represent a comparison of the current Parallax implementation to the open-source Cisco iSCSI initiator[4] connected to a Network Appliance F840 filer. As mentioned above, the focus to date has been on the functional aspects of the design presented in Section 5.1.1 and not on raw throughput. The implementation of course also supports live migration— currently a `cvd_sync()` must be explicitly issued prior to the final relocation to ensure correctness.

Deletion

Two command-line utilities are provided to allow the reclamation of storage space. `cvd_snap_delete` may be used to delete a range of snapshots from an existing disk, while `cvd_delete` deletes the disk itself. Both deletion tools use the radix tree scanning algorithm described in Section 5.1.1, and are able to reclaim data at a rate of 8MB/s concurrently on each storage brick in the replica group. Snapshot deletions do not remove pages in the snapshot log; they remain for the life of the CVD. Snapshot log pages are, however, a negligible overhead on storage consumption.

5.1.2.5 *Implications of Copy-on-Write*

It is worth considering the consequences of using copy-on-write-based snapshots for regular OS installations. One concern with regard this issue is that of scheduled system tasks making daily updates to the file system. In particular, `updatedb`—the indexing invocation of `slocate`—is typically run once per night, and builds an index of all files in the file system.

A concern in the use of block-level CoW is that the file access-time (`atime`) updates caused by a whole-disk scan will result in considerable copy-on-write overhead. Figure 5.6 shows the block write overhead that result from running `updatedb` on a 1.3GB Fedora Core install above a variety of com-

[4] `http://linux-iscsi.sourceforge.net/`

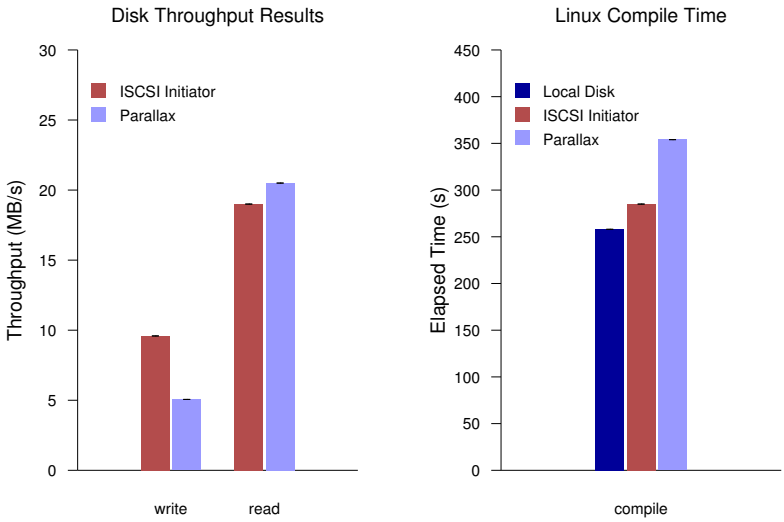


Figure 5.5: Throughput and Compile Performance

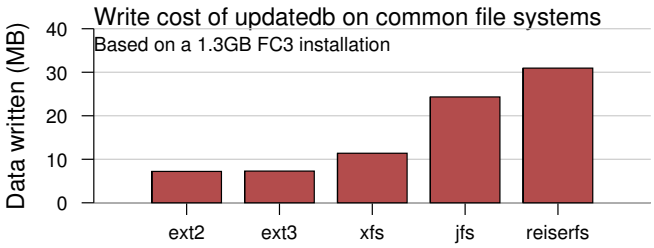


Figure 5.6: Write Cost of `updatedb` on File Fystems

mon Linux file systems. The overheads range from seven to about thirty megabytes per day, which should be acceptable in most systems. Alternatively, these overheads may be avoided by mounting the file system with an option preventing access-time updates.

5.1.3 Future Work

The prototype implementation demonstrates the efficacy of the Parallax design design and provides a useful working environment. It also suggests several avenues for future work, a few of which are described here.

5.1.3.1 *Combining Duplicate Blocks*

Parallax provides a block-level interface to distributed storage that offers private virtual disks, with the assumption that most users will use a conventional file system over a CVD to provide a higher level storage interface. The design has focused on aggressive copy-on-write mechanisms across snapshot histories and between templates and their derivative images to share data between VMs that need not even be aware of each other.

This type of implicit sharing can be extended even further to combine additional redundant blocks that arise for reasons other than common CVD ancestry. These reasons may include similar package installations on different CVDs, duplicate files within a file system, or blocks of all zeros used by an application to reserve space in a file. It is clearly worth investigating the extent to which such serendipitously shared data accrues in typical deployment scenarios. The following two methods would provide the ability to detect and exploit these duplicate blocks:

Block Janitor

The first potential extension is a background *block janitor* process that searches for duplicate blocks stored in multiple locations in the store. Its operation would be similar in spirit to the log cleaner process proposed in log structured file systems [RO92]. The janitor will find duplicate read-only blocks, link all related radix trees to one of the duplicates, and delete the others.

Such an operation would interfere with the existing delete mechanism; since links to the duplicated block would all have to be read-only to prevent incorrect deletion, the existing delete mechanism would never be able to delete such a block. The situation where blocks are independently created in multiple locations and then all instances are removed is not likely to be common, however, and would likely be outweighed by the storage gains from collapsing duplicates.

Content-based Block Map

An alternative to the janitor is to re-implement the distributed block store as a a service mapping content hashes to global block addresses, permitting content-based addressing in place of the direct block lookups presently sup-

ported. With this service in place, writes can still be made initially to the local persistent cache and a content hash can be computed asynchronously between the time the VM write operation completes and the block is flushed to the block store. The content-hash map is then consulted, and a new block is allocated only when the block does not already exist in the block store.

Slow operations like content hashing and collision detection are kept out of the critical performance path, and read operations are completely unaffected. As with the janitor, links to duplicate blocks must all be read-only and may introduce block leaks under the current deleting rules.

5.1.3.2 *Security*

While security is a clear concern for storage systems, its discussion has been avoided in the text for fear of unnecessarily complicating the functional description of the design and implementation. The threat model on which the Parallax design is based extends that of most other OS virtualization work: Specifically, it assumes that the underlying virtualization software is protected by a set of narrow and clearly-defined interfaces from the overlying virtual machines. VMs using CVDs have access only to the virtual block addresses of the device, and not to the global block addresses used by the distributed block store.

Arbitrary levels of security may be engineered into a cluster where Parallax is deployed through combinations of isolating the storage VMs and block store servers from client VMs using techniques such as network partitioning, traffic filtering, encryption and authentication. For the most part though, a hardened implementation is left as further work.

A secondary concern in the security of the current implementation is the resource consumption risk. A malicious VM could potentially consume large amounts of disk data, starving other VMs. The performance starvation has been addressed by virtual machine research in the past. Ensuring that VMs cannot consume arbitrary disk space can be enforced by placing allocation limits on individual CVDs. All allocations made in a CVD may be counted, and when full old snapshots may have to be deleted to free space. This has not been implemented in the current work, but should be rather straightforward.

5.1.4 Current Status

The Parallax work has attracted considerable interest, and development will continue following the submission of this thesis. I am actively working with the UCSD honeyfarm effort to incorporate the use of Parallax into their system, and will also be working towards a Parallax deployment as a basis for VM hosting on development and test machines in the Computer Laboratory over the coming year.

5.2 Summary

The deployment of VMMs in cluster environments presents many new challenges in the management of storage resources. This chapter has presented the Parallax storage system as a device service which attempts to address these issues. Parallax enlists the use of storage VMs as device extensions on each physical host within a cluster. The Parallax server is a soft device-based extension that runs in this VM and provides cluster virtual disks to client VMs. The storage VM itself allows the benefits of device services presented in Chapter 3; storage is unified into a single administrative slice within the cluster and may be managed independently of other facilities. The Parallax work has been one of the most fruitful aspects of this thesis, as it has evolved into a larger development effort which will continue in the lab over the following year.

While the Parallax work represents the most complex block device extension that has been constructed to date, it is hardly the only one. Earlier local-host extensions included simple copy-on-write and encrypted block devices. Additionally, a user-level version of the block backend driver has been constructed, and was used as the source for the block throughput results described in Chapter 3. The user-level block backend is currently being deployed in production systems to allow access to virtual file systems stored in image files on network filers.

The next chapter discusses a final set of examples of device extensions. It explores opportunities for soft devices to be applied along with modifications to other aspects of virtual hardware in order to build sweeping architectural extensions in development systems.

Chapter 6

Supporting System-Wide Architectural Change

This chapter introduces a final set of examples of the application of device extensions. The work presented here demonstrates that, in combination with other techniques, device extensions may be used to introduce sweeping architectural changes to a system, allowing developers to experiment with new system-wide hardware features. Two specific examples are presented:

1. **Device Support for Pervasive Debugging.** Pervasive debugging (PDB) involves the use of virtualization to enable debugging both *vertically* through the entire software stack including OS and application software, and *horizontally* across a set of virtual hosts. PDB effectively adds a distributed, system-wide debugging layer with new virtual hardware features such as physical address-based watch points and distributed breakpoints. Device extensions are used to extend debug triggers down onto devices themselves, allowing debug entry based on request addresses and data.

2. **Device Support for Taint-based Data Isolation.** Network-attached computers are under constant threat of exploitation, and are protected using various forms of firewall and intrusion detection. A major challenge in this form of admission-based protection is in discerning good traffic from bad. Rather than attempting to block specific traffic, taint-based isolation attempts to provide the invariant that data received from the network may never be executed. Device extensions are used to mark tainted data as it moves in and out of the system from network and disk, while additional techniques are employed to track the

propagation of tainted data at an instruction granularity.

Both of the applications described in this section have resulted from efforts to apply device extensions to other research efforts in the lab. This chapter provides general descriptions of these projects and focuses on the device-level aspects of the work that represent my own contributions. In addition to providing additional examples of the techniques presented throughout this thesis, I believe that the collaborative nature of these investigations illustrate the immediate benefit of low-level device extensions, and their applicability to other research projects.

6.1 Device Support for Pervasive Debugging

In recent years, virtualization has been revisited as a useful basis for research in debugging. Two notable new approaches have been the the use of virtual machine replay to allow "time-travel" debugging — extending the debugger to step execution backwards as well as forwards in time [KDC05] — and the use of virtual machines to debug large systems, entire OS instances and distributed systems that span multiple OSes, by running a set of hosts together on a common physical machine [HH05].

The pervasive debugging (PDB) project at the Computer Laboratory addresses the second of these two issues. It is specifically interested in using VMM-based debugging extensions to broaden the *scope* of traditional debugging tools. PDB extends scope in two ways: *Vertical Debugging* allows control flow to be followed across layers of a system, through OS, application run-time, and application code. Conversely, *Horizontal Debugging* allows a distributed system composed of a set of virtual hosts to be run together above the debugger, and facilitates the understanding of complex error conditions that are often present in such systems.

6.1.1 Hardware Modifications in PDB

Existing debuggers take advantage of hardware support for debugging. The Intel x86 architecture, for instance, provides a set of four debug registers which allow breakpoint and watchpoints to be set on executing code and

data. Virtual addresses are installed in these registers, and flags are set to activate them. In the case of an instruction breakpoint, the processor will generate a fault when an attempt is made to execute code at the specified address. In the case of a watchpoint, the processor will trap on attempts to access data at the specified address. In addition to break- and watchpoints, the x86 provides support for single-stepping, in which a debug trap is generated after the execution of each instruction.

The approach taken by PDB can be viewed as adding new hardware support for debugging within a system. This is done to a degree within non-virtualized OSes to cope with the limited number of debug registers. If a developer attempts to set more breakpoints than the number of debug registers supports, an OS will typically provide support in one of two ways: First, it may enable constant single-stepped execution and examine each instruction in an attempt to emulate breakpoints in software. Second, and more efficiently, it may use memory protection in combination with this sort of emulation to mark pages containing break- or watchpoints to force traps whenever data on those pages is accessed, at which point the debugger may test accesses. This technique will be revisited, as it forms the basis of the second example in this chapter, taint-based data isolation.

PDB's addition of hardware support extends these techniques by taking advantage of virtualization to add debug features below the OS – effectively in virtual hardware. Two examples of this are modifications to the system to allow break- and watchpoints to be physically addressed, and applied to a distributed system. In the interest of brevity, the discussion that follows refers to both break- and watchpoints simply as breakpoints.

Physical address-based breakpoints allow conventional breakpoints to be applied to physical as well as virtual addresses within a system. This functionality is especially useful in OS development and in debugging applications which use shared-memory communications, where physical memory may become aliased across a collection of virtual addresses. Using shadow page tables [Wal02, Gum83, HR91], a VMM-based debugger may validate newly created page table entries, treating them specially if there is a breakpoint on the underlying physical page.

A second example of a virtual hardware capability added by PDB is that of distributed breakpoints and distributed single-stepping. With these tools, a

breakpoint may trigger the suspension of a collection of executing virtual machines, and their progress may be followed in detail as a group. This feature is obviously useful in isolating bugs in distributes systems and in diagnosing system-wide failure conditions.

6.1.2 Soft Device Support for Debugging

Existing application debuggers such as `gdb`[1] are powerful development tools both for isolating bugs, and for understanding how software functions. Above the hardware debug facilities mentioned above, debuggers such as `gdb` provide a very simple interface, allowing six operations on an application: The debugger may read and write values in memory, it may read and write registers, it may step to the next instruction, and it may resume normal execution. Additionally, a set of operations are provided to establish break- and watchpoints.

PDB extends the existing debug interfaces to provide the horizontal and virtual-scoped debugging described above. However, debugging is still limited to interactions with CPU and memory. To address this, I have constructed soft device-based extensions to allow breakpoints on interactions with I/O devices. For instance, the debugger may be triggered whenever a new packet arrives from the network. While this is possible without the aid of virtualization, for instance by placing a breakpoint at the top of an OS's network interrupt handler, it requires sufficient understanding of the OS code to identify where this handler is. By adding device debugging support to a VMM-based debugger, I/O paths may be debugged in an OS agnostic fashion. Moreover, more complicated debugger activations, for instance filtering for a specific type of packet, may be implemented once in the isolated extension and applied to all OSes being debugged.

Unlike conventional watchpoints, which are specified according to memory addresses in the debug target, specifying device-based watchpoints requires a rather broader interface and is specific to a class of device. I have implemented device watchpoints as soft devices for both network and disk.

Network watchpoints are implemented using a modified version of `snort-`

[1] `gdb` is available at `http://www.gnu.org/software/gdb`

`inline`[2], a popular intrusion detection program. Snort includes a packet specification language that allows events to be triggered on matching packets. The implementation allows the debugger to push new packet-matching rules down to the network soft device, and will trigger debug entry on their arrival.

Disk watchpoints use a slightly simpler interface, allowing the specification of blocks based either on virtual block address ranges, or on block contents. Block content matches are specified using regular expressions, and when activated, all blocks passing to and from a VM are scanned and an event is generated on a match.

In both cases, the debug event is sent to the debugger and the device request is queued. This allows the debugger to do any additional interrogation or modification on the request message and VM before the message is delivered. Common actions are to switch the VM into single step mode or modify page permissions on the request page, and then allow the debug device to deliver the request.

6.1.3 Understanding Intrusions

By using the IDS rules that are included with `snort`, the network debugging extension may be used to trigger debugger entry on the arrival of packets containing system exploits. The resulting tool allows the debugger to be used to trace arbitrary exploits from the moment that they arrive at the OS interrupt handler, through to the point at which the system is compromised, and forms a simple but useful illustration of this approach.

In addition to this, the device extensions for debugging may be used to perform fault injection and event logging. In general, the addition of device extensions for debugging allow a flexible tool to facilitate complex debugging tasks. In addition to the simple examples here, device extensions may be applied in a cluster environment to trigger debugger entry on access to a file used in a shared file system, or on transmission of particular network packets. The example in the next section extends the techniques developed in the work on device support for debugging to introduce a new system feature that spans disk, network, memory, and CPU behaviour.

[2] `snort-inline` is available at `http://snort-inline.sourceforge.net/`

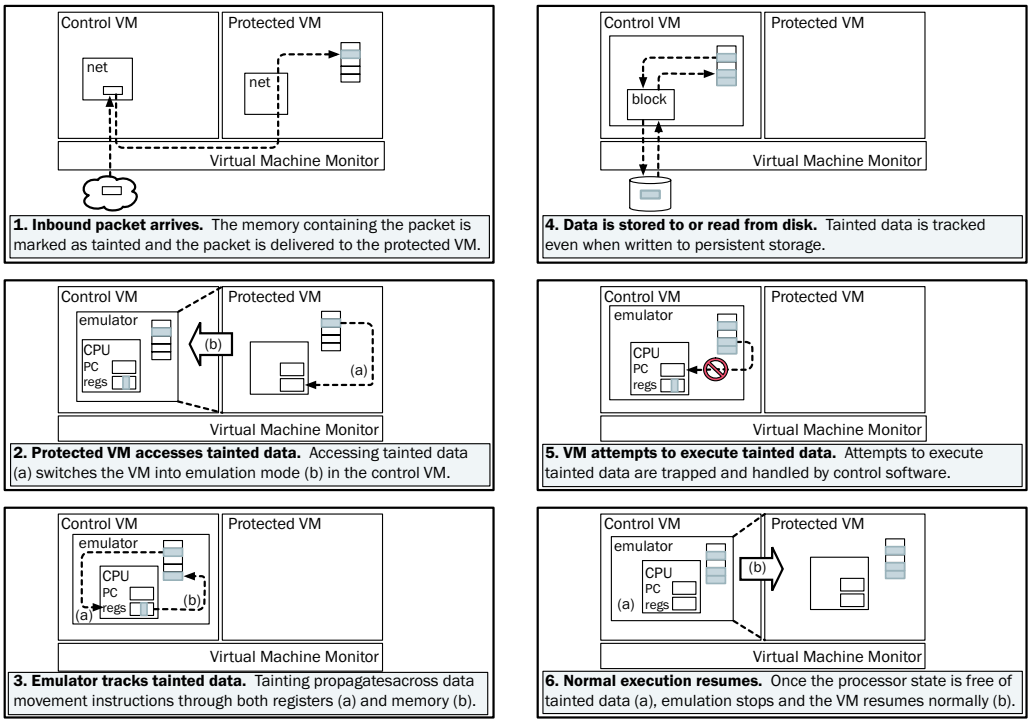


Figure 6.1: Overview of Taint Tracking

6.2 Taint-based Data Isolation

Most existing approaches to protecting computers from malicious software involve the use of firewalls and intrusion detection systems (IDSs) to distinguish good traffic from malicious traffic. This is a complicated task, and is increasingly prone to outbreaks in which an exploit propagates faster than the signature or firewall rule required to protect against it. As an alternative, we chose to investigate a modification to the system in which network data would be prevented from being executed on a host. By mandating that data from the network be immutably non-executable, a system is protected against the large class of exploits in which a host is tricked into running malicious software.

6.2.1 Overview of Taint-tracking

Figure 6.1 illustrates how tainted data is managed as it moves through the system. In (1), a packet arrives at the host from the physical network inter-

face. The packet is processed in the control VM, which is hosting the network device and the pages into which the data has been received are marked as tainted as they are transfered in to the protected VM's pseudo-physical address space.

Shortly later, in (2), the OS in the protected VM handles the interrupt indicating that a new packet has been delivered. As it processes the packet headers, the CPU attempts to load the tainted memory pages. This load results in a fault, that causes the VM to be switched into emulation. The current CPU state is transferred into a hardware emulator running in user space of the control VM, where execution continues. The emulator has direct mapped access to the VM's memory, and so continues to execute the machine in place.

As this execution progresses, as shown in (3), emulated instructions operate on tainted data. The emulated processor microcode is modified to track tainted data across memory and register stores. Additionally, stores of untainted data result in the cleaning of tainted memory. In this manner, well-behaved OSes will clean tainted memory as they zero freed pages.

(4) shows that tainted data is also tracked across storage to disk. The virtual disk is modified to store an on-disk data structure mapping the location of tainted data, as data is reread the memory it is placed in is marked appropriately.

If a VM attempts to execute a piece of tainted memory, as in (5), either by loading the address of tainted pages into EIP or advancing execution into a tainted region of memory, an instruction exception is thrown. This exception can use an existing processor fault, such as invalid opcode, or we may extend the processor to add a tainted execution-specific exception. In an effort to avoid modifying the hardware interface, we opt for the former of these two options in our work.

Finally, after processing a region of tainted data, the processor finds that its registers have become clean. At this point, it executes to an appropriate opportunity, and then transfers the processor state back to virtualized execution.

6.2.2 Device Extensions

Tainted data is tracked across network and block device interfaces by implementing taint-tracking soft devices. This approach is particularly useful in this situation, as I/O-related data may be marked and tracked outside the scope of the protected virtual machine.

6.2.2.1 *Tainting Network Data*

As described in Chapter 3, Xen's current network interface delivers packets to individual VMs using page remapping. Inbound packets are received into a system-wide pool of free pages, which have been "donated" by VMs that use the network. A packet is written into an empty page, routed to the destination VM, and delivered to that VM by mapping the page into its physical address space, in place of a previously donated page.

The taint-tracking device extension marks all received data on network devices as tainted. Page table entries containing tainted data are then marked as not-present, resulting in a trap when a VM attempts to access them, after which VM access may be analyzed in emulation until CPU registers are free of tainting.

Processing packet headers results in considerable time spent in emulation, and involves code that is generally well tested and trusted to be safe. As a performance optimization, we allow the virtual network interface to be configured to deliver untainted headers. In addition to mapping the packet's page to the receiving VM, we send a copy of the headers in shared memory, which the protected VM may process without incurring taint faults.

6.2.2.2 *Tainting Storage*

The virtual storage interface in Xen is structured similarly to that of the network, but does not involve page exchanging as the target memory for reads and writes always belongs to the protected VM and is known at the time of request. The virtual disk device, running in the control VM, maintains a persistent data structure – a sparse tree – identifying all disk blocks that have been marked as tainted. On writes, the taint properties of memory are preserved on disk. An additional benefit to this data structure is that it is not part of the virtual disk, and so cannot be addressed by the protected VM.

On reads from disk, the target pages are marked as read-only for the protected VM while the request is serviced, memory is remarked as tainted wherever necessary, and then the data is made available.

The current implementation of taint tracking is not directly integrated with Parallax. This integration will be pursued as future work, as the mapping structures used by Parallax provide an ideal location to store taint annotations.

6.2.2.3 *Selective Tainting*

In some situations, it may be desirable to strike a further compromise between the performance overhead of taint tracking and the thoroughness of the system. As an extension of the header exemptions described for the taint-tracking virtual network interface above, `snort` may be used to monitor all traffic that will be delivered to the protected VM, as described in regard to device extensions for debugging.

This approach allows an interesting extension to the functionality of the IDS, in which it may divide inbound traffic into three classes: Attack traffic, for which IDS rules already exist, may simply be dropped as they normally would by such an application. A white list may then identify traffic that need not be tainted, for instance from trusted hosts or to particular trusted services. Finally, remaining traffic may be treated as untrusted and passed into the system with taint-based protection. One benefit to this technique is that it allows administrators to focus on the set of traffic that they do trust, rather than that which they do not, but does not require that unanticipated traffic be dropped—it is just treated with an enhanced degree of precaution.

6.3 Summary

This chapter has described an additional set of examples of device extensions. In these examples, device extensions are coupled with modifications and extensions to other aspects of the virtualized system, including memory modifications, using shadow page tables, and changes to CPU behaviour, by dynamically switching into emulated execution.

Using these techniques, it is possible to experiment with sweeping new hardware features. In one example, a VMM-based debugger is extended to allow debugger entry to be triggered off of certain device accesses. In another, device extensions are used to track tainted data received from the network, and the system is then modified to guard against its eventual execution.

Both of the examples here demonstrate not only the usefulness of device extensions in a VMM environment, but also the potential for their application to broader systems features. Both of the projects described in this chapter are recent, and ongoing work within the lab.

Chapter 7

Related Work

Prior to concluding, this chapter places the work presented in this thesis in the context of relevant related research. The work just presented covers a broad range of systems topics, and the related work is consequentially very broad. Prior work relating to network denial-of-service prevention as relating to the symmetry-based traffic limiter was described in Chapter 4. Similarly, relevant work on distributed storage pertaining to Parallax was included in the presentation in Chapter 5.

This chapter aims to survey the broader research areas into which split devices, device extensions, and device services directly apply. It begins with a summary of the history of virtual machine monitors and then moves on to survey the treatment of both access to device hardware and software extensibility within operating systems. It ends by mentioning a sample of device extensions that have been developed by other researchers, and which could be applied using the techniques presented in the preceding chapters.

With regards the thesis of this work, a key insight that this chapter attempts to expose is that of the design decision in computer systems development that underpins the desire to provide extensibility in the first place. As with many other systems features, the ability to cleanly extend any system involves the introduction of a layer of indirection that necessarily carries with it a performance overhead. The evolution of operating systems for personal computers has been punctuated by the gradual adoption of techniques, previously used on mainframes, which improved the dependability and manageability of the system at some cost in performance. Multiple protection domains, multi-processing, shared libraries, and most recently hardware virtualization are all examples of this evolution. The survey of related work

presented in this chapter aims to demonstrate that where the performance overhead of device extensions is acceptable, that the approaches demonstrated in this thesis represent a powerful technique to develop and deploy device-level extensions in modern systems.

7.1 Virtual Machines

A virtual machine monitor (VMM), or *hypervisor*, is a narrow layer of software that presents a set of isolated execution environments that closely match the underlying physical computer. Each of these environments is called a virtual machine (VM), and may contain an operating system and associated set of applications.

Virtual machines have existed since the 1960s, primarily as an approach to achieving improved utilization on expensive mainframes. Goldberg's 1974 article in IEEE Computer, "Survey of Virtual Machine Research" [Gol74] cites over 70 papers relating to VM research and development at that time. The paper also identifies and discusses many problems relating to virtualization such as issues in dealing with nonvirtualizable hardware, and techniques to reduce virtualization overhead.

The paper also discusses many applications of virtual machines that hold true today. It discusses the use of VMs to host legacy applications that are "locked" to a specific operating system, to facilitate OS development and testing, and to test and debug networks and distributed systems by virtualizing them on a single host. IBM's VM/370 [SM79] was a VMM used on IBM mainframes to achieve exactly these goals.

After a long period of relative inactivity, virtualization returned to systems research in the 1990s with the Stanford work on Disco [BDGR97b, BDGR97a], and later Cellular Disco [GTHR99]. Disco used virtualization to run commodity operating systems on non-commodity large-scale, shared-memory multiprocessors: the work targeted NUMA machines and was initially based on hardware simulation. They observed that adding support to existing OSes would be prohibitively complex, and that virtualization presented a viable approach to making use of the resources on such systems. Disco showed that virtual machines were a useful and effective ab-

straction for managing resources and providing hardware fault isolation on SMP hosts.

In recent years, the performance of commodity x86-based hardware has improved sufficiently that virtualization has become practical. Several commercial packages have become available both for desktop (e.g. VMware Workstation, Connectix Virtual PC), and server (e.g. VMware GSX/ESX Server, Virtual Iron) environments. Workstation products have taken the approach of providing a hosted VMM, which runs as an application above the native OS. Typically this does not require modifications to the existing host, but consequently results in high performance overhead. VMware's server products run on the "bare metal", and are able to somewhat reduce performance overheads. In all cases, the commercial virtualization offerings are driven by the desire to host unmodified operating system binaries, due to the need to transparently support existing software.

Unfortunately, the x86 platform does not provide direct support for virtualization. Virtualizing the x86 hardware requires additional effort from a VMM, as discussed in Section 2.2. To safely virtualize the x86 for unmodified binaries, a VMM must perform run-time transformations of the executing code, rewriting difficult-to-virtualize operations and replacing them with appropriate calls. An alternative approach is to sacrifice support for unmodified OS binaries, and make modifications to OS code that ease the interactions between the OS and the VMM. Dubbed *paravirtualization* [WSG02], the approach of adapting the VMM-OS interface has been used throughout the history of virtualization to improve performance, and accommodate difficult-to-virtualize architectures [You73, Gol74].

The influx of research into virtualization on the x86 platform has shown considerable results over the past few years. Two significant research efforts into building paravirtualizing hypervisors have been undertaken: Denali [WSG02] initially aimed to host thousands of virtual machines on a single physical host as a platform for hosting Internet services. Xen [BDF+03] was based on a goal of providing strong, resource-accounted partitioning within the host and aiming to accountably divide a system between a smaller number (i.e. 100) of VMs. These two projects, in conjunction with existing virtualization and emulation packages that have become available for the x86 architecture (e.g. User-mode Linux [Dik00], Qemu [Bel05]),

have led to a resurgence of investigations into the *applications* of virtual machines. In aggregate, this research has demonstrated that not only are virtual machines a useful tool for resource isolation and management, but that they may be beneficially applied to tasks such as intrusion detection and forensics [DKC+02, JX04, VMC+05], configuration management and debugging [WCG04], software debugging [KDC05, HH05], and OS migration [HJ04, CFH+05]. Moreover, virtual machines have formed the basis for many proposals to build distributed computing platforms [HHKP03, AR02, GC04].

Interestingly, the administrative benefits that drove interest in virtualization in the 60s and 70s have been partly responsible for the recent revitalization in the research area. VMMs have become attractive as a potential solution to the management problems faced in large computing facilities such as clusters and data centers [KUS+04, JX03]. The narrow layer of control software provided by a VMM allows the separation of the traditional system administrator role into two parts: A *facilities administrator* who manages physical machines, performing repairs and service to hardware, and a *software administrator*, who manages the OS and applications installed within an individual virtual machine. Many commercial ventures (e.g. VMware, XenSource, Virtual Iron) currently provide software and support for the use of VMs to manages large system installations.

7.2 Device Access in Operating Systems

The structure of device drive code within operating systems has been a long-standing area of investigation in OS construction. While efficient drivers are critical to the I/O performance of a host, drivers are a frequent source of bugs and consequentially of system instability. Presumably due to performance concerns, most current commodity OSes include driver code in the same protection domain and address space as the kernel itself, and are victim to severe crashes in the face of driver failure.

Drivers have been long-held as a common source of bugs in OS code. Static analysis of the Linux 2.4.1 kernel for a set of relatively simple programming bugs, such as calling a blocking function with interrupts disabled, found

that 85% of bugs occurred within device driver code [CYC+01]. Many approaches have been taken to isolate driver code in order to mitigate the severity of crashes. Interestingly though, driver dependability has only recently occurred as motivation for user-level drivers.

While microkernels such as Mach [BRS+85] typically include device drivers within the kernel itself, several projects have explored extracting drivers into their own protected processes. Mach 3.0 introduced "Device Independent Drivers", in an attempt to eliminate repeated code common to most drivers. They specified low-level device interfaces for common device classes (e.g. SCSI, Ethernet) and provided a minimal in-kernel driver to map a particular device to this interface. Generic driver code then ran as a Mach process, interacting with this interface over IPC [FGB91]. This work was extended by Golub *et al* to allow multiple drivers to share multiplexed access to a given physical device through the support of a resource manager in the kernel [GSI93].

Several other microkernel systems have explored user-level device drivers. The Raven system provided a minimal microkernel for hosting tasks on a multiprocessor. The kernel provided simple interrupt dispatching, allowing hardware drivers to run completely in user space [RN93]. The motivation behind Raven's structure was that by delivering hardware interrupts directly to drivers, the overhead of both context switches and data movement across the kernel-user boundary could be avoided. Similarly, QNX is a microkernel-based system, largely targeting embedded devices, that allows user-level driver tasks to register for hardware interrupts and claims improved performance as a result [Hil92]. U-Net [vEBBV95] presented a narrow, virtualized ATM interface which could be driven by applications in SunOS and was also motivated by the performance benefits of managing hardware directly from the context of the client application.

Like Xen, Nemesis [LMB+96] was concerned with both isolation and accounting of applications. By removing drivers from the kernel and providing a narrow dispatch layer, drivers could be executed in user space. Pratt's "User-Safe Device Architecture" claims that drivers may be made both safer and accountable using this approach. Arsenic [PF01], demonstrated an application of these techniques to a programmable network interface and served as groundwork for the techniques used to manage devices in Xen.

Almost all of these approaches have been specifically concerned with moving driver code into user-space in developmental, often microkernel-based systems. More recently, several projects have applied this technique to isolate "unmodified" device drivers. In particular, the Fluke kernel incorporated drivers from several modern OSes which were wrapped with glue interfaces from the OSKit [FBB+97]. This represents initial work motivated by the pragmatic concerns of system dependability and software maintenance: Using existing unmodified drivers was seen as a necessary approach to keep up with the rapidly evolving hardware market. The Exokernel also incorporated some OSKit drivers, while L4Linux incorporated Linux drivers above the L4 microkernel [HHLS97].

Over the past few years, these approaches have also been applied to commodity kernels with the specific intention of improving system reliability. Rather than running drivers directly as user-space applications, unmodified drivers are isolated a hardware protection domain, but run as-if they were still in-kernel. Nooks [SBL03] uses such an approach within Linux to isolate existing drivers. this is achieved by instrumenting the rather ambiguous Linux driver interface to force a context switch, and carefully account resource usage by drivers so that memory may be returned to the system if a driver crashes and is restarted. Both L4 [J. 04,LUSG04] and Xen [FHN+04] allow unmodified drivers to run in isolated virtual machines, avoiding the complexities of finding a cut-point through complex in-kernel driver interfaces. The approaches taken by both of these efforts are fairly similar, and to my knowledge the work was carried out simultaneously by both groups independent of one another.

In addition to supporting legacy drivers in isolated execution, several recent efforts are reexamining the benefits of allowing driver development and execution in user-space, both on academic microkernels [EG04], and on Linux [Chu04]. These follow similar motivation to that of earlier work on driver proxies for Windows NT, which observed that driver writing was a difficult and error-prone process and that by redirecting NT I/O request packets (IRPs) to userspace, driver construction could be facilitated [Hun97].

In addition to this work, there have been industrial efforts to set a common interface to device drivers that may be applied across OSes. The Uniform Driver Interface UDI [UDI99] was developed by a consortium of both

hardware and OS vendors in an effort to enable the development of stable and portable device drivers. A common driver interface is a laudable effort, and one which could easily be embraced by the work in this thesis. Unfortunately, the project appears to have stagnated, as there has been no new information posted to the UDI web site[1] since 2001. The consortium did produce a set of specifications which appeared to suffer from "interface unioning" across the stakeholders: the idealized interfaces used in Xen have taken the opposite approach, opting for simple, narrow interfaces to shared devices. The position taken by this thesis is that in order for devices to be shared in a VMM there must be *some* interface between device and client VMs, and that it is in the interest of developers to keep this interface relatively stable over time. Given that position, the techniques described in this work may be applied to whatever low-level interface prevails.

The grant table-based approach to sharing memory among VMs, which is described in Chapter 3 draws on a wealth of cross-domain memory sharing work in the existing literature, most notably mechanisms such as those described in Fbufs [DP93] and IOlite [PDZ00]. The grant table approach extends that taken to cross-domain memory sharing in the L4 microkernel, and uses very similar terminology. Somewhat confusingly, an L4 *grant* is equivalent to a page transfer in Xen, while L4's *map* operation corresponds to a page grant in Xen. The L4 and Xen techniques are similar in that the place the core components of memory sharing within the most privileged part of the system. However, their use is rather different: L4's operations are tightly coupled with external pagers and memory sharing is exposed at a high-level within the system, for instance to construct file systems. In its current form, memory sharing in Xen is used almost exclusively for device-level communications. Broadening the scope of shared memory use within the VMM must be carefully considered as it represents a risk of increasing the amount of cross-domain shared-state, which in turn may compromise the benefits of isolation.

The extension work in this thesis follows the motivation of isolating extensions for safety and ease of development set out by previous work. By working at the driver interface presented to VMs, extensions as described in this thesis achieve isolation while remaining applicable across systems and give the extension developer complete freedom with respect to the environ-

[1] `http://www.projectudi.org/`

ment (OS, language, tools) that they work in.

7.3 Extensibility in System Software

Extensibility — the insertion of extension code into a running operating system — is a well explored area. Vino [SESS96] and SPIN [BSP+95] both used language extensions and trusted compilers to generate and sign extension code, and incorporated safe run-times within the OS to attempt to contain misbehaving extensions. Exokernel [EFO95] and Nemesis [LMB+96] provided minimal near-physical interfaces in an effort to allow applications the freedom to make low-level changes that extensions are used for in other systems. The structure of these systems is similar to virtual machine monitors in that a small kernel provides physical resource sharing and isolation between coarse-grained protection domains.

Interposition on OS interfaces is also a long-standing approach to adding functionality to systems. The extensibility-focused OSes above provided OS support for extensions on OS-internal interfaces. SLIC [GPRA98] allows calls to kernel interfaces such as system calls and signals to be intercepted and redirected to extensions running in either kernel or user-mode. Their approach used modifications to kernel binaries to allow extensions to be added without modification to a kernel's source. Similarly, several kernel instrumentation packages such as DTrace [CSL04] and KernInst [TM99] use binary patching to redirect OS execution at a function granularity. While these approaches allow arbitrary OS extension, assuming access to the OS symbol table, they provide little or no safety: extensions effectively have free reign to access system resources and extension crashes can result in complete system failure.

In addition to the mechanistic issues involved in actually applying extensions to an OS kernel are the practical concerns with the portability and maintenance of extensions. Most commodity OSes now provide some form of support for inserting extension code into the OS kernel. However, extensions are generally specific to the OS version that they were written for. Unlike the application binary interface (ABI), which is held static to ensure that applications may run across versions of an OS, extensions are tightly

coupled with internal OS interfaces, which are highly specific to an individual OS, and which change over time. [FGCW05] discusses the difficulties of maintaining extensions that interact directly with OS source over time.

7.4 Device Virtualization and Extension

The SPIN work was later modified to investigate the installation of "device extensions" on to so-called smart hardware – I/O devices with embedded processors capable of running software within the device. In this work, dubbed SPINE [FMBC98], the extensible aspects of the SPIN kernel were moved to Windows NT 4.0 and support was written for a Myrinet programmable network card. Extensions, written in MODULA-3 could be installed on the NIC, allowing certain I/O operations to be carried out independent of the CPU. SPINE's extension model aimed to provide *performance* improvements for devices: the authors present a video decoder that DMAs incoming frames directly to the frame buffer, and a IP router that DMAs packets across a pair of network interfaces.

Throughout their history, virtual machines monitors have explored two main approaches to handling I/O devices, centered around the design decision of whether to preserve the device's hardware interface or to present an alternate virtualized interface. Preserving the interface of existing hardware, as described with regard to VMware workstation in [SVL01], has the single clear benefit that VM-based OSes may use existing drivers. In this approach, the VMM captures device requests and passes them to some form of a device emulator which translates them and issues them to the real device. While [SVL01] presents several possible performance improvements, the approach is bound to involve a degree of inefficiency, as a logical request (e.g. sending a packet) involves the emulation of a large number of instructions, each of which requite a transition out of the running VM. even so, the benefit of supporting unmodified OS binaries may often be worth this overhead, especially when modifying the OS is not possible.

An alternate approach is to explicitly modify device interfaces to support virtualization. This technique, which has been used frequently [BDGR97b, KEG+97,WSG02,BDF+03], presents an *idealized* device interface to the VM

which eschews the complexity and overhead of emulation.

Work on Virtual Channel Processors (VCPs) [MN03] has argued for the use of virtual machines to take better advantage of hardware parallelism for I/O-related computation. Referring to the performance benefits that have been achieved by other systems in offloading tasks such as protocol processing onto a separate CPU [MS98, MS00, RBB+02], the VCP work proposes placing I/O subsystems in isolated VMs and scheduling them on dedicated CPUs. The basis of their argument is that the economies of scale that exist for general purpose processors allows such CPUs to increase in performance at a higher rate than processors on embedded devices. Similar to Disco, this work argues for the use of VMs to take advantage of multiprocessors, but at a sub-OS granularity.

μDenali's proposal for "interposable virtual hardware" [WCSG04] is likely the most similar work to that described in this thesis. Both approaches argue for the use of interposition on VM device interfaces to implement extensions. The μDenali work takes a high-level view of virtual hardware, and presents an initial argument for the validity of the approach while focusing on the need for a standardized interface to VMM operations. As described in Chapter 2, Denali is a considerably less mature VMM, allowing the multiplexing of single-threaded and single-address space "operating systems" that have been linked against Ilwaco, a BSD derivative kernel library. Their work adopts IPC mechanisms similar to those in Mach [BRS+85] to transport device requests, and provides mechanisms for VMs to interpose on those transports. They argue that extensions should be provided with a standard API extending beyond device interfaces to include issues such as VM execution control (suspend/resume), access to VMM state, and to the internals of VM state.

The work presented in this thesis extends the ideas introduced in [WCSG04] by considering them in a mature VMM and applying them to real-world OSes. Soft devices achieve considerably better performance, achieving results that make extensions practical to use in production environments. This thesis additionally focuses on the development and management of practical extensions, with particular interest in cluster environments.

7.5 Smart Devices

Applications of software extensions behind device interfaces are plentiful. Notable examples include block extensions for secure [GNA+97], versioning [WCG04], distributed [LT96,SFV+04], and intrusion-detecting [PSG+03] disks. Network devices have been extended to provide packet filtering, rate limiting and admission control [EK96,PF01], protocol scrubbing [MWJH00], and intrusion detection [WCSG04].

Extensions are frequently unportable, as they are developed either for a specific OS or behind a network protocol which may not be supported by all OSes. Existing work demonstrates a wealth of exciting ideas for device extensions, only a sample of which are described here. VMM-based device interfaces and the approaches described in this thesis allow considerably more structure and support for the development of these extensions, and facilitate practical deployment on commodity systems.

Chapter 8

Conclusion

In concluding the presentation of this thesis, this section discusses opportunities for further investigation, and summarizes the overall work.

8.1 Future Work

Several aspects of this thesis remain open to additional investigation. This section summarizes several opportunities for future work based on soft devices.

8.1.1 Extending the Existing Extensions

Three of the extensions described in this thesis have led to additional research interest in the lab, and investigation into them is continuing beyond the course of this thesis.

The Parallax storage service presented in Chapter 5 has received considerable interest and we are hoping to work toward a complete prototype which will be used to manage disks on test boxes used for Xen development. It is my hope that further development will lead to deployments in production environments.

As mentioned earlier, the symmetry-based rate limiting work discussed in Chapter 4 is an aspect of a larger investigation into symmetry-based limiting. With Christian Kreibich, Jon Crowcroft, Steven Hand, and Ian Pratt, I intend to work toward a deployment of a stand-alone symmetry-based lim-

iter for deployment in a college network. We hope to explore the impact of such a limiter on real traffic, in order to further validate the approach.

The taint-based memory protection presented in Chapter 6 is ongoing work. We hope to work with the authors of the Vigilante [CCC+05] system to combine our techniques for the protection of commodity system. In addition, the dynamic emulation described in brief in that chapter is a promising approach for a variety of interesting system extensions which will certainly continue.

8.1.2 Other Device Interfaces

In addition to building additional extensions based on the current interfaces, it will be worth while to apply these techniques to other device types. As mentioned in Chapter 3, work is currently underway in the Computer Lab to build a file system level split device driver. Additionally, it may be interesting to explore interposition-based extensions for video or audio devices.

8.2 Summary

This thesis has argued that an architecture based on virtual machine monitors may be used to build device extensions that are safe, flexible, and achieve reasonable performance. Chapters 1 and 2 framed the problem, and provided an overview of the Xen virtual machine monitor.

In Chapter 3, I described a set of techniques for the construction, extension, and aggregation of virtual devices and discussed the implementation of these approaches for disk and network extensions for VMs running on Xen. The development of extensions, or *soft devices*, was facilitated by device taps—drivers which allowed extensions to interpose on the device channels used by virtual machines. Measurement results were presented showing that soft devices were of sufficiently low overhead to be practical for all disk and many network applications.

I additionally introduced the notion of aggregating soft devices to form *device services*. Such services allow the I/O function provided over a specific I/O interface, storage for instance, to be isolated as a separate entity within

a cluster environment. This approach allowed such services to be isolated in terms of performance, failure, security, and administration.

Chapter 4 explored extensions for network device management. It presented an approach to preventing virtual machines in hosting facilities from being exploited to perform denial of service attacks. The approach enforced a limit on traffic symmetry, the ratio of transmitted to received packets, to throttle the transmission capabilities of VMs who do not appear to be receiving acknowledgement traffic from the hosts they are transmitting to.

As a second example area in which these techniques could be applied, Chapter 5 presented applications of these extensions within the domain of storage. It described Parallax, a storage system for clusters of virtual machines that allowed fast and frequent snapshotting, and efficiently stored large numbers of virtual disks based on a collection of common images.

Chapter 6 explored opportunities to combine device extensions with broader changes to a VMM-based system. An initial example was to to support debugging, where the introduction of device-based watch points allowed a debugger to be activated based on the contents of device data. The chapter went on to discuss the development of taint-based memory protection, in which device extensions are combined with dynamic emulation to introduce a sweeping new virtual hardware feature requiring changes to CPU, memory, and device behaviour.

In summary, this thesis has demonstrated that the mechanisms provided by virtual machine monitors may be combined to build a compelling framework for device extensions. I believe that this approach to both extending individual devices, and treating device functionality as a service within a cluster of physical hosts is a useful approach to constructing system software, and one which will gain considerable use in years to come.

Bibliography

[ALPS03] Daniel Adkins, Karthik Lakshminarayanan, Adrian Perrig, and Ion Stoica. Taming IP Packet Flooding Attacks. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003. 85

[And04] S. Andersen. Changes to functionality in Microsoft Windows service pack 2, Part 2: Network protection technologies. `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx`, August 2004. 83

[AR02] Amr Awadallah and Mendel Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW 2002)*, August 2002. 126

[ARW03] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003. 85

[BALL90] B.Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990. 66

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM SOSP*, pages 164–177, October 2003. 10, 73, 74, 78, 125, 131

[BDGR97a] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997. 124

[BDGR97b] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 143–156, October 1997. 19, 35, 37, 124, 131

[Bel99] Steven M. Bellovin. Distributed firewalls. In *;login:*, pages 39–47, November 1999. 83

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings USENIX Annual Technical Conference*, pages 41–46, 2005. 125

[BRS+85] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications. *IEEE Software*, 2(4):65–67, July 1985. 127, 132

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995. 130

[CCC+05] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, 2005. 135

[CFH+05] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. USENIX Symposium on Networked Systems Design and Implementation*, 2005. 25, 81, 126

[CG05] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proc. USENIX Annual Technical Conference*, pages 379–382, 2005. 66

[Chu04] P. Chubb. Get more device drivers out of the kernel! In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, July 2004. 32, 128

[CSL04] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual*

Technical Conference, General Track, pages 15–28, 2004. 130

[CYC+01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, October 2001. 1, 127

[Dik00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000. 125

[DKC+02] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, December 2002. 10, 126

[DP93] P. Druschel and L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 189–202, New York, NY, USA, 1993. ACM Press. 129

[EFO95] D. Engler, Kaashoek F, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995. 35, 130

[EG04] K. Elphinstone and S. Götz. Initial evaluation of a user-level device driver framework. In *Proceedings of the Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, Beijing, China, Sept 2004. 32, 128

[EK96] D. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 53–59, Stanford, California, August 1996. 1, 80, 133

[FBB+97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997. 1, 30, 128

[FGB91] A. Forin, D. Golub, and B. Bershad. An I/O system for mach

3.0. In *Proceedings of the Second USENIX Mach Symposium*, pages 163–176, Monterey, CA, Nov 1991. 127

[FGCW05] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, New Mexico, June 2005. 30, 131

[FHN+04] K Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS-1)*, October 2004. 27, 75, 128

[FMBC98] M. Fiuczynski, R. Martin, B. Bershad, and D. Culler. SPINE: An operating system for intelligent network adapters. Technical Report TR-98-08-01, University of Washington Department of Computer Science, 1998. 131

[FS00] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, IETF, May 2000. 83

[GC04] S. Goyal and J. Carter. Safely harnessing wide area surrogate computing - or how to avoid building the perfect platform for network attacks. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, December 2004. 126

[GNA+97] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–284, 1997. 1, 133

[Gol72] R. Goldberg. Architectural principles for virtual computer systems. PhD thesis, Harvard University, 1972. 8, 14

[Gol74] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, June 1974. 9, 10, 124, 125

[GPRA98] D. P. Ghormley, D. P. Petrou, S. H. Rodgigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference*, pages 39–52, June 1998. 130

[GSI93] D. Golub, G. Sotomayor, and F. Rawson III. An architecture

for device drivers executing as user-level tasks. In *Proceedings of the Third USENIX Mach Symposium*, pages 153–171, Santa Fe, NM, Apr 1993. 127

[GTHR99] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 154–169, December 1999. 124

[Gum83] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983. 21, 115

[Hea03] D. Heap. Taurus: A taxonomy of the actual utilization of real unix and windows servers, Jan 2003. White paper. 10

[HG04] Mark Handley and Adam Greenhalgh. Steps Towards a DoS-resistant Internet Architecture. In *Proceedings of the Second ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA-04)*, pages 49–56, 2004. 85

[HH05] A. Ho and S. Hand. On the design of a pervasive debugger. In *Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging*, Monterey, California, Sept 2005. 10, 114, 126

[HHKP03] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform. In *Proceedings of the 6th OPENARCH*, April 2003. 126

[HHLS97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of micro-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 66–77, Oct 1997. 128

[Hil92] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126. USENIX Assoc., 1992. 127

[HJ04] J. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004. 126

[HP91] N. Hutchinson and L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991. 61

[HR91] Judith S. Hall and Paul T. Robinson. Virtualizing the VAX Architecture. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 380–389, 1991. 21, 115

[Hun97] Galen Hunt. Creating user-mode device drivers with a proxy. In *Proceedings of the 1st USENIX Windows NT Workshop*, Seattle, WA, Aug 1997. 32, 128

[HWF+05] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, New Mexico, June 2005. 12, 66

[J. 04] J. LeVassuer and V. Uhlig. A Sledgehammer Approach to Reuse of Legacy Device Drivers. In *Proceeedings of the 11th ACM SIGOPS European Workshop, Leuven, Belgium*, September 2004. 128

[Ji05] M. Ji. Instant snapshots in a federated array of bricks. Technical Report HPL-2005-15, HP Laboratories, 2005. 96

[JX03] X. Jiang and D. Xu. Soda: A service-on-demand architecture for application service hosting utility platforms. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 174, Washington, DC, USA, 2003. IEEE Computer Society. 126

[JX04] X. Jiang and D. Xu. Collapsar: A vm-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, pages 15–28, 2004. 10, 93, 126

[KC03] S. T. King and P. M. Chen. Backtracking intrusions. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 223–236, 2003. 93

[KDC05] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference*, 2005. 10, 91, 93, 114, 126

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Granger, Hèctor M. Briceño, Russel Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth

Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, October 1997. 131

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000. 57

[KUS+04] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. Softudc: A software-based data center for utility computing. *Computer*, 37(11):38–46, 2004. 126

[KWC+05] C. Kreibich, A. Warfield, J. Crowcroft, S. Hand, and I. Pratt. Using packet symmetry to curtail malicious traffic. In *Proceddings of the Fourth Workshop on Hot Topics in Networks (HOTNETS)*, 2005. 63, 79, 84, 88, 90

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, December 1993. 66

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. 127, 130

[LT96] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, 1996. 1, 96, 133

[LUSG04] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004. 12, 128

[Mic04] Microsoft. Transcript: Windows XP service pack 2. `http://www.microsoft.com/windowsxp/expertzone/chats/transcripts/04sept01.mspx`, September 2004. 83

[MN03] D. McAuley and R. Neugebauer. A case for virtual channel

processors. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 237–242, 2003. 132

[MP96] D. Mosberger and L. Peterson. Making paths explicit in the scout operating system. In *Operating Systems Design and Implementation*, pages 153–167, 1996. 61

[MS98] Steve Muir and Jonathan Smith. AsyMOS – An Asymmetric Multiprocessor Operating System. In *Proceedings of OPENARCH '98*, April 1998. 132

[MS00] Steve Muir and Jonathan Smith. Piglet: A Low-Intrusion Vertical Operating System. Technical Report MS-CIS-00-04, University of Pennsylvania, January 2000. 132

[MST+05] A. Menon, J. Renato Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM Press. 66, 67

[MWJH00] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In *Proceedings of IEEE INFOCOM-00*, pages 1381–1390, 2000. 133

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000. 129

[PF01] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proceedings of IEEE INFOCOM-01*, pages 67–76, April 2001. 1, 80, 127, 133

[PG74] G. Popek and R. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974. 14, 15

[PP03] S. Personick and C. Patterson, editors. *Critical Information Infrastructure Protection and the Law: An Overview of Key Issues*. National Academies Press, 2003. 82

[PSG+03] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the USENIX Security Symposium.*, August 2003. 133

[QD02] S. Quinlan and S. Dorward. Venti: A new approach to archival

storage. In *Proc. 1st USENIX Conference on File and Storage Technologies, Monterey, CA*, January 2002. 104

[QTSZ05] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM Press. 91

[Rad01] M. Radin. Distributed denial of service attacks: Who pays?, 2001. Whitepaper commissioned by Mazu Networks. 82

[RBB+02] M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel. Tcp servers: Offloading tcp/ip processing in internet servers. Technical Report DCS-TR-481, Rutgers University, Department of Computer Science, March 2002. 132

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA*, pages 129–144, August 2000. 14

[RN93] D. S. Ritchie and G. Neufeld. User level IPC and device management in the Raven kernel. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 111–125, San Diego, CA, Sept 1993. 127

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992. 110

[Ros94] T. Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, 28(4):48–55, 1994. 61

[SBC+03] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003. 75

[SBL03] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM SOSP*, pages 207–222, October 2003. 1, 30, 128

[Sca01] S. Scalet. See you in court. *CIO Magazine*, November 21 2001. 82

[SESS96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, October 1996. 130

[SFH+99] D. Santry, M. Feely, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 110–123, 1999. 104

[SFV+04] Yasushi Saito, Svend Frølund, Alistair C. Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *ASPLOS*, pages 48–58, 2004. 1, 96, 133

[SM79] L. Seawright and R. MacKinnon. VM/370 – a study of multiplicity and usefulness. *IBM Systems Journal*, pages 4–17, 1979. 124

[SPLS+02] A. C. Snoeren, C. Partridge, C. E. Jones L.A. Sanchez, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM Transactions on Networking (TON)*, 10:721–734, Dec. 2002. 85

[SVL01] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association. 34, 131

[SWKA01] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Network support for IP traceback. *IEEE/ACM Transactions on Networking (TON)*, 9(3):226–237, Jun. 2001. 85

[TM99] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999. 130

[UDI99] Introduction to UDI version 1.0. Project UDI, 1999. Technical white paper, *http://www.projectudi.org/*. 30, 128

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. In *Proceedings of the fifteenth ACM symposium on operating systems principles*, pages 40–53, Copper Mountain, CO, 1995. 127

[VMC+05] M. Vrable, J. Ma, J. Chen, E. Vandekieftand A. Snoeren D. Moore, G. Voelker, and S. Savage. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, Brighton, UK, October 2005. 10, 93, 126

[Wal02] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002. 21, 37, 115

[WCG04] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, pages 77–90, 2004. 1, 10, 61, 91, 93, 126, 133

[WCSG04] A. Whitaker, R. Cox, M. Shaw, and S. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 169–182, March 2004. 1, 12, 30, 61, 132, 133

[WFHD05] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proceedings USENIX Annual Technical Conference*, pages 379–382, 2005. 58

[Whe00] D. A. Wheeler. Estimating linux's size. `http://www.dwheeler.com/sloc/`, November 2000. 33

[WHTP02] A. Warfield, S. Hand, T.Harris, and I. Pratt. Isolation of shared network resources in Xenoservers. Technical Report PDN-02-006, Planet Lab Design Note, November 2002. 48, 63, 78

[Wil02] Matthew M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the 18th Annual Computer Security Applications Conference (ADSAC 2002)*, December 2002. 83

[WRF+05] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, New Mexico, June 2005. 75, 92

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002. 16, 35, 125, 131

[You73] C. J. Young. Extended architecture and hypervisor performance. In *Proceedings of the ACM SIGARCH-SIGOPS workshop on virtual computer systems*, pages 177–183, 1973. 125

[YPS03] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against ddos attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 1–15, 2003. 85