IMPROVING VIRTUAL HARDWARE INTERFACES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ben Pfaff
October 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mendel Rosenblum) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dan Boneh)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Monica Lam)

Approved for the University Committee on Graduate Studies.

Abstract

Since the origin of virtual machine monitors in the 1960s, virtual hardware has often been designed with "impure" interfaces different from physical hardware interfaces. Paravirtualization, as this is now termed, is often used to simplify VMMs and boost VM performance. This thesis explores tradeoffs in a rarely seen form of paravirtual interface, where the virtual interface operates at a higher level of abstraction than the common hardware interface. We in particular examine the effects of providing a BSD socket-like interface to a VM instead of an Ethernet interface, and the effects of providing a file system interface instead of a block device interface.

Our experiments show that higher-level ("extreme") paravirtualization has direct benefits for sharing, security, and modularity. We also show that our approach requires little or no change to existing VMMs and operating systems. Given the availability of processor cores, it has minimal performance cost: 3% or less in every case for our paravirtual network stack, and under 4% for file system macro-benchmarks. In special cases, we even demonstrate speed-ups.

We further extend the extreme paravirtualization approach for virtual storage. Virtual disks have many attractive properties, such as a simple, powerful versioning model and an architecture that makes it easy to create and economically store large numbers of VMs. They also suffer from serious shortcomings: low-level isolation prevents shared access to storage, versioning takes place at the granularity of a whole virtual disk, and lack of structure obstructs searching or retrieving data. An extreme paravirtualization interface to storage aids sharing, but not their other shortcomings.

Therefore, this thesis proposes the concept of a *virtualization aware file system* (VAFS) that combines the features of a virtual disk with those of a distributed file system. A VAFS

extends a conventional distributed file system with versioning, access control, and disconnected operation features resembling those available from virtual disks. This gains the benefits of virtual disks, without compromising usability, security, or ease of management.

Acknowledgements

Many people deserve thanks for spurring the ideas that make up this thesis. I thank my advisor, Mendel Rosenblum, for encouraging me to develop my ideas over my six years at Stanford, Dan Boneh and Monica Lam as my readers, Kunle Olukotun as chair of my orals committee, and John Mitchell and Phil Levis as additional committee members.

Tal Garfinkel contributed extensively to discussions of the ideas behind this work, especially regarding virtualization aware file systems.

Martin Casado, Jim Chow, Matt Eccleston, Steve Hand, Joe Little, Tim Mann, Richard McDougall, Pratap Subrahmanyam, Jeremy Sugerman, Paul Twohey, Carl Waldspurger, Junfeng Yang, and the NSDI and SOSP anonymous reviewers all supplied valuable feedback for this thesis and its antecedents.

Mark Bellon, Minoru Massaki, Thomas Richter, Tim Shoppa, Melinda Varian, Tom Van Vleck, Dave Wade, Lynn Wheeler, Gio Wiederhold, Linda Yamamoto, and the Computer History Museum contributed greatly to the historical research presented in Chapter 2.

Elizabeth Stinson and Charles Orgish supplied technical assistance in setting up the PON and POFS experiments.

Finally, I owe my wife a great deal of gratitude for her patience and support over the years it took me to conceive and execute this work.

This work was supported in part by the National Science Foundation under award 0121481 and by TRUST (Team for Research in Ubiquitous Secure Technology), which also receives support from the National Science Foundation under award CCF-0424422.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Virtual machine monitors (VMMs) are in increasingly common use. We envision a time when, with few exceptions, all desktop and server systems will run on top of a VMM. As virtual environments rise to ubiquity, the virtual hardware interfaces between a VMM and each operating system on top of it in a virtual machine (VMs) become at least as important as the physical hardware interfaces accessed by the VMM. Virtual interfaces may even have greater importance than physical interfaces: only a single VMM interacts with physical hardware at a given time, but that VMM may simultaneously host a large number of VMs.

From the origin of VMMs in the 1960s, the virtual hardware interfaces provided by many VMMs have been specialized to the virtual environment. These kinds of specialization, called *paravirtualization*, have most commonly aimed to simplify VMM implementation, to improve performance or reduce resource requirements, or to improve the user experience. The current resurgence of VMMs has also employed specialized virtual hardware for all of these reasons.

Empirically, these interface specializations, especially those for modern operating systems, tend to have three properties in common. First, they tend to be visible only to operating systems, not to application programs, due to VMM authors' strong motivations to maintain compatibility with existing applications. Second, they tend to limit the extent of needed changes so that they fit within an existing interface layer in OS kernels, to ensure clean integration with operating systems. Finally, in most cases these specialized virtual

interfaces are designed by streamlining, simplifying, or subsetting a common physical interface. For example, rather than simulating an Ethernet controller with an interface for a legacy I/O bus, VMMs such as Xen provide a streamlined virtual Ethernet interface with shared memory transmit and receive rings.

This thesis investigates a more rarely seen form of virtual interface specialization that shares the first two of these properties—application compatibility and limited changes to operating systems—but not the third. In particular, the virtual interfaces we propose do not resemble common physical interfaces, but instead operate at a higher level of abstraction. The direct benefits of these higher-level interfaces include manageability, security, and improved potential for sharing. Regarding manageability, a device module can be repaired or replaced separately from the VM that it serves, among other benefits. As for security, interposing at a higher level allows us to move all the code implementing the lower level parts of the virtual device into a new domain of trust, which allows us to better secure a system or a network, or to shrink the trusted code base of an operating system. Finally, by pulling lower-level code out of the operating system into another virtualization layer, we enable that layer to serve multiple client VMs. This enables sharing of resources such as file systems and network connections among VMs in ways not easily achievable previously. (When isolation is a priority, resources need not be shared.)

Our higher-level virtual interfaces also increase the modularity of operating systems. The resulting increase in modularity reaps the benefits traditionally claimed by other modular OS approaches, e.g. microkernels [1], such as increasing the operating system's flexibility, extensibility, maintainability, security, and robustness. Unlike traditional microkernels, the new structure falls out from an evolutionary approach that permits a gradual migration from existing monolithic operating systems to a more modular, decomposed operating system.

This thesis studies two kinds of higher-level virtual interfaces, which we call "extreme paravirtualization":

- Virtual storage, which is usually provided in the form of a virtual hard disk. We study supplying the VM a higher level of abstraction, that of a virtual file system. The resulting storage is easier to share, administer, search, and revise.

- Virtual networking, which is usually provided in the form of a virtual Ethernet interface. We study supplying the VM a higher level interface, that of a virtual TCP/IP stack. This yields benefits in sharing, security, and modularity.

File systems and networking stacks are large pieces of code, often much larger than a simple paravirtualizing virtual machine monitor. Thus, instead of integrating them into the VMM, we implemented each one in a separate, isolated *service virtual machine*. Our prototype service VMs in turn implement the desired functionality in terms of the customary lower-level virtual interface: a virtual Ethernet interface for networking, a virtual disk for storage.

This thesis further extends the extreme paravirtualization approach in the case of virtual storage. On their own, virtual disks have many attractive properties as the basis of VM storage, such as a simple, powerful versioning model and an architecture that makes it easy to create and economically store large numbers of VMs. Unfortunately, they also suffer from serious shortcomings: their low-level isolation prevents shared access to storage, versioning takes place at the granularity of a whole virtual disk, and their lack of structure obstructs searching or retrieving data in their version histories. A higher-level virtual interface to storage aids sharing virtual disks, but it does not help with their other shortcomings.

Therefore, this thesis proposes the concept of a *virtualization aware file system* (VAFS) that combines the features of a virtual disk with those of a distributed file system. A VAFS extends a conventional distributed file system with versioning, access control, and disconnected operation features resembling those available from virtual disks. This gains the benefits of virtual disks, without compromising usability, security, or ease of management.

Difficulty in implementation or poor performance could be seen as strikes against extreme paravirtualization, but neither is a serious issue in practice. Implementation is eased by the way that modern operating systems are designed to support multiple implementations of components. We simply implemented our own components that work by communicating with a service VM. In fact, no kernel changes at all were required to support our network and file system prototype implementations under Linux, only addition of our kernel modules. As for performance, it is aided by the advent of multi-core and multi-threaded CPUs, which provide shared caches that are useful for high-speed communication among

VMs. The speed penalty is thus minimal even in our relatively unoptimized implementation: 3% or less for our network stack and under 4% for file system macro-benchmarks. In special cases, we even demonstrate speed-ups.

1.1 Contributions

This thesis makes the following contributions:

- We propose that virtual machine systems offer virtual interfaces at a significantly higher level of abstraction than physical interfaces. We call this *extreme paravirtualization*.
- We detail benefits of extreme paravirtualization, such as fine-grained sharing of services among VMs, increased security, and increased modularity. In turn, increased modularity has potential for improved flexibility, extensibility, maintainability, security, and robustness.
- We show that extreme paravirtualization can be implemented for network stacks and file systems in existing virtual machine monitors and operating systems with little or no change to either. We demonstrate that the performance penalty is under 4% for our prototype implementations.
- We speculate on use of extreme paravirtualization as a primitive to gradually evolve monolithic OSes into a more modular form that offers the same benefits as other modular OS structures, such as microkernels. We show how such an evolution could avoid a difficult transition for users and administrators at any single point.
- We further propose a model for virtual storage, called a *virtualization aware file system* (VAFS). A VAFS combines the powerful versioning features of virtual disks with the fine-grained structure and access control of distributed file systems.
- We explore trade-offs in VAFS design. The hierarchical nature of VM versioning informs a file system design with hierarchically versioned files and directories. Differing expectations for VM and distributed file system access control lead to multiple

orthogonal forms of access control. We also describe the need for and implementation strategy for disconnected operation.

1.2 Terminology

We adopt the following terminology for virtual machines. A virtual operating system running inside a virtual machine is a *guest OS*. When a VMM runs on top of a general-purpose operating system, that operating system is the *host OS*.

1.3 Organization

The remainder of this thesis is organized as follows. The following chapter chronicles the history of virtualization from its inception, with emphasis on the history of interfaces to virtual hardware, and contrasts these interfaces with those proposed by this thesis. Chapter 3 motivates our work. Chapter 4 describes our proposed extreme paravirtualization interfaces for networking and file system access and implementation details of our prototypes for each device module. Chapter 5 details our proposal for virtualization aware file systems, including Ventana, our VAFS prototype. In Chapter 6, we evaluate performance and other aspects of our prototypes. Chapter 7 presents related and future work, and Chapter 8 concludes.

Chapter 2

Virtual Interfaces

This chapter chronicles the history of virtualization from its inception, with emphasis on the history of interfaces to virtual hardware, and contrasts these interfaces with those proposed by this thesis.

2.1 Roots of Virtualization in Time-Sharing

The invention of the virtual machine monitor in the 1960s can be seen as a logical consequence of trends in evolution of computers and their use that began in the 1950s. Computers in the 1950s were designed to run only a single program at a time. Until the invention of interrupts in 1956, in fact, overlapping computation with I/O often required the programmer to carefully break computations into code segments whose length matched the speed of the I/O hardware [2]. This can be further seen in that, until approximately 1960, the term *time-sharing* meant multiprogramming, or simply overlapping computation with I/O [3, 4, 5].

A single-user architecture meant that each computer's time had to be allotted to users according to some policy. Two very different policy models dominated [6, 7]. In the first, known as the *open shop* or *interactive* model, users received exclusive use of a computer for a period of time (often signed up for in advance). Any bugs in the user's program could then be fixed immediately upon discovery, but the computer was idle during user "think time." Furthermore, the number of users who could use a computer during a fixed amount of time increased only minimally as the speed and size of the computer increased. Faster

and larger computers were also more expensive and therefore on these machines the user overhead had a higher opportunity cost. Hence only the smaller computers of the era tended to be available interactively.

In the contrasting *closed shop* or *batch* model, users prepared their job requests off-line and added them to an off-line queue. The machine was fed a new program from the queue, by its professional operators, as soon as it completed the previous one. Batch processing kept the machine running at its maximum capacity and scaled well with improvements to the machine's speed. The difficulty of debugging, however, increased because the turnaround time from job submission to completion was typically measured in hours or days [8]. An article about virtual machines in 1970 contrasted the two models this way [9]:

> Remember the bad old days when you could sit at the console and develop programs without being bothered by a horde of time-hungry types? Then things got worse and they closed the door and either you took a 24 or 48 hour turnaround, or they let you have 15 minutes at 1:15 AM on Sunday night.

The weaknesses in both models became increasingly apparent as computer design progressed to faster and larger machines. Demand for computer time tended to grow faster than its supply, so efficient use of computer time became paramount. Moreover, the increasing size of the problems that could be solved by computer led to larger, more complex programs, which required extensive debugging and further increased demand [5]. These factors pushed in contradictory directions: increasing demand called for the increased machine efficiency of the batch model, but human efficiency in debugging required on-line interaction [6].

Additionally, on the horizon were proposed uses of computers for experiments in interactive teaching and learning, computation as a public utility, "man-computer symbiosis," and other forms of "man-machine interaction" [5, 10, 6, 11, 12]. Some believed as early as 1962 that a time would come when access to a computer would be universally important [13]:

> We can look forward to the time when any student from grade school through graduate school who doesn't get two hours a day at the console will be considered intellectually deprived—and will not like it.

These future needs could not be fit into either model. A new way was needed.

Interactive time-sharing was the answer. It achieved human efficiency, by providing a response time for program debugging and other purposes on the order of seconds or minutes instead of hours, as well as machine efficiency, by sustaining the machine's CPU and I/O utilization at or near their limits.

Once time-sharing became the goal, the next question was how to design the user interface for these new time-sharing systems. To anyone of the era who had had the opportunity to use a machine interactively, the obvious answer was that it should look as though the user had a computer to himself. The early discussions of time-sharing systems emphasized this aspect. For example, in a 1962 lecture, John McCarthy described the goal of time-sharing as: "From the user's point of view, the solution clearly is to have a private computer" [6]. Similarly, in an MIT report proposing research into time-sharing systems, Herbert Teager described its goal as presenting "...all the characteristics of a user's own personal computer..." [14].

This orientation naturally carried over to early time-sharing system implementations. The authors of the APEX time-sharing system built in 1964, for example, said that it "simulates an apparent computer for each console" [15]. A time-sharing system at UCB was described in a 1965 paper as built on the principle that "...each user should be given, in effect, a machine of his own with all the flexibility, but onerousness, inherent in a 'bare' machine" [12]. These systems were not exceptional cases, as reported in a 1967 theoretical treatment of time-sharing systems [10]: "Time-shared systems are often designed with the intent of appearing to a user as his personal processor."

It should not be surprising, then, that many of these early time-sharing systems were *almost* virtual machine monitors. The APEX system mentioned above, which ran on the TX-2 machine at MIT, is representative. Its "apparent computers" were described as "somewhat restricted replicas of TX-2 augmented by features provided through the executive program." The restrictions included a reduced amount of memory and removal of input/output instructions, for which the executive (kernel) provided equivalents through what are called system calls today. (Much later, R. J. Creasy is reported to have said about one of these systems that they were "close enough to a virtual machine system to show that 'close enough' did not count" [16].)

Another time-sharing system of this type was M44, a machine developed at IBM's Yorktown Research Center between 1964 and 1967 [17, 18, 19]. It was based on an IBM 7044 machine, whose hardware was modified to increase the size of the address space and add support for paging and protection. In the first known use of the term *virtual* in computing [19], the M44 simulated a "a more or less ideal computer, or *virtual* machine closely related to the M44," which they called 44X. The M44/44X system was not, however, any closer to a true virtual machine system than the other time-sharing systems of its day: the 44X was sufficiently different from both the IBM 7044 and the M44 that no existing software ran on it without porting. M44/44X is thus notable for its introduction of terminology, but it was not a virtual machine system.

2.2 The First Era of Virtualization

A 1974 paper by Popek and Goldberg defines a VMM as software that meets three conditions [20]. First, a VMM must provide an environment essentially identical to that of the machine that it simulates, except for resource availability or timing. Thus, the VMM must isolate the VM from other activities on the same host. Second, the VMM must run the simulated software at near-native speed. Third, the VMM must assert control over all the virtual machine's resources.

By the mid-1960s all the prerequisites for such virtual machine systems were available. Moreover, the researchers working on time-sharing systems were oriented toward creating the illusion of multiple private computers. In retrospect, the invention of the virtual machine monitor seems almost inevitable. Under the Popek definition, the first genuine virtual machine system was CP-40, developed between 1965 and 1967 at IBM's Cambridge Scientific Center [16]. CP-40 was built on top of the then-fledgling IBM System/360 architecture, which was binary and byte-addressed, with a 24-bit address space [21]. System/360 lacked architectural support for virtual memory, so CP-40 was developed on a machine whose hardware was extended with a custom MMU with support for paging [22].

CP-40's virtual hardware conformed to the System/360 architectural specification well enough that it could run existing System/360 operating systems and applications without

modification. Its conformance did have a few minor caveats, e.g. it did not support "self-modifying" forms of I/O requests that were difficult to implement [23]. CP-40 did not initially provide a virtual MMU, because of the amount of extra code required to do so, but a later experimental version did include one [16, 24].

CP-40's immediate successor was CP-67, developed by IBM from 1966 to 1969, also at the Cambridge Scientific Center [16]. Unlike M44 and CP-40, CP-67 ran on unmodified, commercially available hardware, the IBM System/360 Model 67 [23]. Later versions of CP-67 provided to its VMs a virtual MMU with an interface identical to the Model 67's [16].

The development of CP-67 also marked an early shift in the design of virtual interfaces. CP-40's virtual interfaces were designed to faithfully implement the System/360 specifications for real hardware. CP-67, however, intentionally introduced several changes from real hardware into its interfaces. To improve the performance of guest operating systems, CP-67 introduced a "hypercall"-based disk I/O interface for guests that bypassed the standard hardware interface for which simulation was less efficient. To reduce resource requirements, CP-67 allowed read-only pages of storage to be shared among virtual machines [25]. An experimental version also added the ability for guest OSes to map pages on disk directly into pages of memory, reducing the amount of "double paging" [26]. To improve user convenience, CP-67 introduced a feature called "named system IPL," which today we would call checkpointing, to speed up booting of VMs [25]. It also added a so-called "virtual RPQ device" that a guest OS could read to obtain the current date and time [16], so that the user did not need to enter them manually.

IBM continued to develop CP-67, producing VM/370 as its successor in 1972. VM/370 added "pseudo page faults" that allowed guests to run one process while another was waiting for the VMM to swap in a page [27]. It also provided additional hypercalls for accelerated terminal I/O and other functions [28].

Work on virtual machines had started to spread into more diverse environments during the development of VM/370. Many of these new VMMs also adopted specialized interfaces for these new reasons. A modified version of CP-67 developed at Interactive Data Corporation in Massachusetts added hypercalls for accelerated terminal support and mapping of disk pages into memory, among other changes, which reduced the resource requirements of

one guest operating system by a factor of 6 [29]. The GMS hypervisor, developed in 1972 and 1973 at IBM's Grenoble Scientific Center in France, accelerated I/O by intercepting system calls from a guest application to the guest kernel and directly executing them in the hypervisor. The VOS-1 virtual machine monitor, developed in 1973 at Wayne State University, was specialized to support OS/360 as its sole guest OS. Because OS/360 did not use an MMU, the virtual MMU support originally included in VOS-1 was removed to improve performance [30]. (VOS-1 ran as a regular process under the UMMPS supervisor, which ran other user processes as well. It was thus also the first "Type II" (hosted) virtual machine monitor [31].)

2.3 The Second Era of Virtualization

As the 1970s drew to a close, the economics that had dictated the need for users to share a small number of large, expensive systems began to shift. Harold Stone accurately predicted the demise of the first era of virtualization in 1979 [32]:

> . . . costs have changed dramatically. The user can have a real, not virtual, computer for little more than he pays for the time-sharing terminal. The personal computer makes better use of the human resource than does the time-sharing terminal, and so the personal computer is bound to supplant the time-sharing computer as the human resource becomes the most expensive resource in a system.

More briefly, R. A. MacKinnon expressed a similar sentiment the same year [27]: "For virtual machines to become separate real machines seems a logical next step."

Indeed, research into virtualization declined sharply in the 1980s. The on-line ACM Portal finds only 6 papers between 1980 and 1990 that contain the phrase "virtual machine monitor," all published before 1986, the least for any decade from the term's introduction onward.

The corresponding decline of commercial interest in virtualization in the 1980s may be seen from the evolution of Motorola's 680*x*0 line of processors. The 68020 processor, introduced in 1984, included a pair of instructions (`CALLM` and `RTM`) that supported

fine-grained "virtualization rings" to ease VMM implementation [33, 34]. In 1987, its successor, the 68030, did not implement these instructions, although it implemented every other 68020 instruction [35]. Furthermore, Motorola documentation for the 68040 and later 680*x*0 models no longer mentioned virtual machines [36, 37].

Virtualization did continue to be of interest on large mainframe systems, where high cost still demanded high machine efficiency. IBM introduced new versions of System/370 with features to improve performance of VM/370 around 1983 [38]. Also in 1983, NEC released VM/4, a hosted virtual machine monitor system for its ACOS-4 line of mainframes that was designed for high performance [39]. Hitachi and Fujitsu also released mainframe virtualization systems, named VMS and AVM respectively, in or about 1983, but it appears that these systems were described only in Japanese [40, 41].

The late 1990s began a revival of interest in virtualization with the introduction of systems to virtualize the increasingly dominant 80*x*86 architecture. In the commercial sector, the revival was again due to the shifting economics of computing. This time, the problem was too many computers, not too few: "... too many underutilized servers, taking up too much space, consuming too much power, and at the end costing too much money. In addition, this server sprawl become a nightmare for over-worked, under resourced system admins" [42]. Virtualization allowed servers to be consolidated into a smaller number of machines, reducing power and cooling and administrative costs.

In academia, virtualization provided a useful base for many kinds of research, by allowing researchers flexible access to systems at a lower level than was previously convenient, among other reasons. The first academic virtual machine monitor of the new era was Disco, which used virtual machines to extend an operating system (Irix) to run efficiently on a cache-coherent nonuniform memory architecture (CC-NUMA) machine [43]. By running multiple copies of Irix, instead of one machine-wide instance, on such a machine, it obtained many of the benefits of an operating system optimized for CC-NUMA with a significantly reduced cost of implementation.

The first commercial virtual machine monitor of this new generation was VMware Workstation, released in 1999. One caveat was that the 80*x*86 architecture was not classically virtualizable according to Popek's definition, which required that the VMM execute most instructions directly on the CPU without VMM intervention. Instead, Workstation

and other 80*x*86 VMMs must simulate all guest instructions that run at supervisor level. Thus, for the purpose of this thesis, we relax the definition of a VMM to include what Popek calls a *hybrid virtual machine* system: a VMM, except that all instructions that run in the VM's supervisor mode are simulated. (The hybrid approach had earlier been used in a VMM for the PDP-10 [44], among others.)

VMware Workstation offered virtual interfaces compatible with physical hardware. It also offered specialized graphics and network interfaces with improved performance [45]. To improve the convenience of its users, it provided a specialized mouse interface that allows a virtual machine to act much like another window on the user's desktop, instead of requiring the user to explicitly direct keyboard and mouse input to the VM or to the host [46].

Other virtualization systems introduced in the late 1990s and early 2000s also used modified virtual interfaces. For simplicity, the Denali virtualization system modified the memory management and I/O device interface to simplify its implementation [47]. To increase performance, the later Xen project used custom virtual interfaces extensively for memory management and device support [48].

2.4 Virtual Interfaces

The preceding history of virtual machine monitors included descriptions of several interfaces between virtual machine monitors and the software that runs inside them. These virtual interfaces can be classified into two categories: *pure* and *impure* [18]. A pure interface is one that simulates the behavior of physical hardware faithfully enough that existing operating systems (and other software) can run on top of it without modification. Any other interface is impure. Thus, impure interfaces include streamlined or tweaked versions of physical interfaces and interfaces wholly different from any physical hardware interface.

Pure interfaces have software engineering elegance in their favor. They also have the advantage that they can be used by existing software without modification. Impure interfaces, on the other hand, require software, in particular operating systems, to be ported to run on top of them. As we have seen, impure interfaces have still been implemented in many virtual machine monitors, usually for one of four reasons:

- *To simplify VMM implementation:* From a VMM implementor's point of view, hardware interfaces are often too complex, due to issues such as protocol and timing requirements of hardware bus interfaces, backward compatibility, hardware error conditions that can't happen in software implementations, and features included in hardware but rarely if ever used by software.

- *To improve performance:* Some hardware interfaces cannot be efficiently implemented in software. For some hardware, each read or write access requires an expensive trap-and-emulate sequence in software simulation. The 16-color graphics modes supported by IBM's VGA display adapter are an example. In these modes, writing a single pixel requires multiple I/O read and write operations [49] that are easily implemented in hardware but difficult to emulate efficiently.

- *To reduce resource requirements:* Some hardware interfaces have excessive memory requirements in the guest or on the host, compared to interfaces designed for virtualization. Interfaces that require buffering are a common example: when implemented in a pure fashion, these often result in redundant buffering. For example, physical terminal interfaces generally require the operating system to buffer data flow. When such a terminal interface is virtualized, and the virtual terminal is connected to a physical terminal, buffering occurs in both the guest operating system and the VMM, wasting memory and time.

- *To improve the user experience:* Because the VMM exists at a level above any individual virtual machine, sometimes it has information that the VMs do not. When it can provide this information directly to the VMs, without explicit action by the user, it improves the user experience.

Three common attributes stand out from the impure virtual interfaces that we have examined, particularly the ones involving modern operating systems. First, although interfaces and software change, there is a strong motivation to maintain compatibility with application programs. The ability to run existing software is, after all, a hallmark of virtual machine monitors. For the modern OS examples, this means that changes to the OS kernel

are acceptable, provided that most application programs continue to run unmodified on the OS.

Second, given the size, complexity, and market dynamics of modern operating systems such as Linux and Windows, the design of impure interfaces also tends to be constrained by the extent of the changes to the underlying OS kernel. In practice, this means that the changes required for an impure interface must fit within an existing interface in the kernel. By doing this, the VMM layer's owners ensure that advances in other parts of the kernel cleanly integrate with VMM-related changes. This can be seen in the focus on changes at the virtual device level, under the device driver interface. Occasionally, this principle has been important enough that new interface layers have been accepted into kernels by their upstream developers for no other reason than to enable an important impure interface, e.g. the `paravirt-ops` patch to Linux that abstracts MMU operations into a VMM-friendly API.

Third, impure virtual interfaces tend to be designed by streamlining, simplifying, or subsetting a common physical interface. For example, rather than simulating an Ethernet controller with an interface for a legacy I/O bus, VMMs such as Xen provide a streamlined virtual Ethernet interface with shared memory transmit and receive rings.

This thesis investigates interfaces that share the first two properties above—application compatibility and limited OS changes—but not the third, that is, our interfaces do not correspond to those of any common physical hardware. The key differences between common impure virtual interfaces and the ones that we propose are:

- Our interfaces operate at a higher level of abstraction than common physical interfaces.
- Our interfaces allow significant amounts of code to be removed from, or disabled in, operating systems that take advantage of them.
- Our interfaces increase the modularity of operating systems that take advantage of them.
- Our virtual interfaces are implemented in separate virtual machines, as virtual machine subsystems, instead of in the VMM.

The following chapter describes our rationales for extreme paravirtualization.

Chapter 3

Motivation

The virtual hardware interfaces provided by many VMMs have been specialized to the virtual environment, in most cases by streamlining, simplifying, or subsetting a physical interface. This thesis proposes virtual interfaces that do not resemble common physical interfaces, but instead operate at a higher level of abstraction. This allows the code implementing the lower level parts of the virtual device to effectively be pulled out of the virtual machine, into a separate module. This chapter describes our motivations for pulling device implementations out of a virtual machine into a separate module.

3.1 Manageability

Virtual machines create new challenges in manageability and security [50]. First, because virtual machines are easy to create, they tend to proliferate. The amount of work involved in maintaining computers increases at least linearly with the number of computers involved, so just the number of VMs in an organization can swamp system administrators.

A second problem in VMs' manageability is their transience: physical machines tend to be online most of the time, but VMs are often turned on only for a few minutes at a time, then turned off when the immediate task has been accomplished. Transience makes it difficult for system administrators to find and fix VMs that have security problems, especially since software for system administration is usually oriented toward physical machines.

Third, whereas physical machines progress monotonically forward as software executes, the life cycle of a VM resembles a tree: its state can be saved at any time and resumed later, permitting branching in its life cycle. On top of transience, branching adds the possibility that fixes, once applied, can be undone by rolling back to a previous unpatched version. Malware and vulnerable software components can thus be stealthily reintroduced long after it has been "eliminated." Common system administration tools are not designed to take this possibility into account.

Pulling a device implementation out of a VM into a separate domain can address some of these manageability challenges. It should be possible for a device module to be administered, repaired, or replaced separately from the VM or VMs to which it provides service. This can reduce the system administration burden from each of the three causes above. VM proliferation has less impact because administrators can focus their efforts on a set of device modules, each of which is much smaller in size than a complete VM and which, taken as a group, are less much heterogeneous than a comparable group of VMs. Device implementations can be designed for off-line as well as on-line maintenance, reducing the impact of VM transience. Finally, a device module need not roll back in lockstep with the rest of a virtual machine, easing the problem of undoing security patches and bug fixes. Versioning of device implementations can be decoupled from the versioning of the VMs that they service: as long as they share a common interface, any device implementation should be able to interface with any VM.

Another potential manageability benefit from pulling out device implementations is increased uniformity of management. Every OS today, and even different versions of the same OS, has different interfaces for managing its network stack and file system. A network or file system module separate from an OS would offer the same administrative interface regardless of the OS it was attached to, reducing management burden.

3.2 Modularity

Pulling the implementation of a device into a separate protection domain increases the modularity of the operating system. Operating system research has identified several direct and indirect software engineering benefits to increasing the modularity of an operating

system [51, 52, 1, 53, 54]:

Simpler Each module is much simpler than a monolithic kernel. The operating system as a whole is easier to understand because interaction between modules is limited to explicitly defined channels of communication.

More robust Small, isolated VMs are easier to test or to audit by hand than entire operating systems, improving reliability. Failing VMs can be restarted or replaced individually and perhaps automatically.

More secure Isolation means that a security hole in one VM, such as a buffer overflow, does not automatically expose the entire operating system to attack. The trusted computing base (TCB) can be reduced from the entire kernel to a small number of VMs. Other modules need not be completely trusted.

More flexible One implementation of a module can be replaced by another, to provide enhanced functionality, better performance, or other attractive features. Modules can be switched while the system is online.

More manageable Manageability is a useful application for the flexibility of a modular VM-based operating system. Modules can be replaced by implementations that configure themselves with policy set by a site's central administrators, for example, without otherwise affecting operating system functionality.

More maintainable Bugs tend to be isolated to individual modules, reducing the amount of code that can be at fault. Smaller modules are easier to modify.

More distributed Individual modules can be moved to other hosts, when this is desirable, simply by extending their communication channels across the network.

3.3 Sharing

The low-level isolation enforced by the structure of traditional virtual machines frustrates controlled sharing of high-level resources between VMs. By pulling device implementations out of an operating system into a separate device module, we enable that layer to

serve multiple client VMs. Thus, extreme paravirtualization facilitates sharing between VMs running on a host. For example, virtual disks can safely be shared between VMs only in a read-only fashion, because there is no provision for locking or synchronization between the VMs at such a low level, but a protocol at the file level can easily support sharing.

Network protocols can also be used for sharing among virtual machines, e.g. existing network file system protocols can be used among virtual machines as easily as they can be used across a physical network. But special-purpose device modules have a number of advantages over general-purpose network protocols. A device implementor can make use of constructs not available across a network. A file system designed for sharing among VMs can, for example, take advantage of memory physically shared among VMs to increase cache coherency, reduce memory usage, and improve performance, compared to a similar network file system. As for a device module designed for networking, communication to such a module cannot also be based on networking without defeating its own purpose.

On a VMM with multiple virtual machines, a shared network device module provides a nice way of sharing the network connections. Scarcity of IP addresses means that they must often be shared among multiple virtual or physical machines. A common solution at the packet level is a network address translation (NAT), in which each machine is assigned a unique IP address that is only routable within the NATed network. Addresses and ports on packets that pass between the NATed network and external networks are then dynamically translated.

NAT has a number of drawbacks. In its simplest form, NAT breaks many Internet protocols and it does not support incoming connections. Advanced implementations paper over these issues with transparent application-specific gateways and port redirection, but these are stopgap measures that add significant overhead. NAT also cannot translate encrypted protocols (unless the router has the key) and to be fully effective it requires the router to reassemble fragmented packets. In short, NAT breaks end-to-end connectivity.

An extreme paravirtualization network architecture permits the use of NAT, if desired, but it also enables an alternative. A gateway VM can connect any number of VMs to a single IP address. The VMs attached to the gateway can then share the IP address in a natural manner. Any of them can connect from or listen on any port (unless forbidden

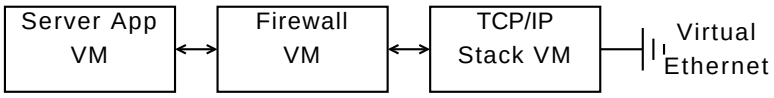

Figure 3.1: VM configuration for increased network security. Boxes represent VMs. Double-headed arrows indicate extreme paravirtualization high-level network links.

by firewall rules), sharing ports as if they were processes within a single VM's operating system.

3.4 Security

Pulling a device implementation into a separate protection domain protects it from some forms of direct attack. For example, an exploit against an unrelated part of the operating system kernel no longer automatically defeats all of the device implementation's attempts at security. Conversely, an exploit against the device implementation no longer defeats all of the kernel's security.

Another form of security benefit applies to enforcement of security policies. In particular, consider network security, in which policies are often implemented by reconstructing a view of high-level activities, such as connections and data transfers, from low-level events such as the arrival of Ethernet frames. Extreme paravirtualization can improve on this situation in at least two ways. First, reconstructions can be inaccurate or ambiguous, due to fundamental properties of the protocols involved [55, 56, 57] or to implementation bugs or limitations [58]. Second, reconstruction has a performance cost that need not be incurred if reconstruction is not necessary.

Extreme paravirtualization also has the potential to reduce the size of the trusted code base (TCB) in some situations. Consider a server VM that contains valuable data. We want to prevent attacks from reaching the VM and to prevent confidential data from leaking out of the VM in the event of a successful attack. In a conventional design, a single VM contains the application, the firewall, and the network stack. A refined conventional design would move the firewall into a separate VM, but both VMs would then contain a full network stack. With extreme paravirtualization, we can refine the design further by pulling the network stack out of both the firewall and application VMs, as shown in Figure 3.1. In this

design, the VMs communicate over a simple protocol at the same level as the BSD sockets interface, such as the PON protocol described in Section 4.2. The firewall VM's code may thereby be greatly simplified. The TCB for the final design is then reduced to the contents of the VMM and the firewall VM, which are both small, simple pieces of code.

In this scenario, the firewall VM has full visibility and control over network traffic. It can therefore perform all the functions of a conventional distributed firewall, even though it does not contain a network stack. It has an advantage over "personal firewall" software, etc., that malware in the application VM cannot disable it, as can happen under Windows [59], on which even Microsoft admits malware is common [60].

The TCP/IP stack VM in this scenario, if compromised, can attempt to attack the external network through the virtual Ethernet or to attack or deny service to the server VM through the gateway VM. However, this is no worse than the situation before the TCP/IP stack is pulled out. In fact the situation is considerably improved in that the TCP/IP stack no longer has access to the server application's confidential data.

The ability to insert a simple "firewall" between an application VM and a device implementation module can also be useful in a file system. This layer could encrypt and decrypt data flowing each way, add a layer of access control, etc. Compared to a virtual disk implementation, it would be able to work at the granularity of a file instead of an entire disk. Compared to a network file system implementation, it could have a significantly reduced trusted code base size, because the interposition layer would have much less code than a typical disk- or network-based file system, as well as better performance (as we will show in Section 6.1.3).

3.5 Performance

Extreme paravirtualization may improve performance because of its potential to reduce the number of layers traversed by access to a device. In a OS in a VM under conventional paravirtualization, for example, file access traverses the OS's file system and its block device stack, then it traverses a similar block device stack in the VMM. Extreme paravirtualization gives us the opportunity to reduce the amount of layering: a sufficiently secure device module could be trusted by the VMM to access hardware directly, eliminating a level of

indirection and potentially improving performance.

Separating a device implementation from the OS makes it easy to replace it by a specialized and therefore possibly faster implementation. For example, Section 6.1.2 shows the performance benefits of bypassing TCP/IP code in favor of a simpler shared-memory based protocol, when virtual machines on the same host communicate. Compared even to the highly optimized Linux TCP/IP networking code, the shared-memory implementation achieves significantly higher performance.

A virtual network stack can also forward the communication to TCP/IP acceleration hardware such as a TCP offload engine. If this reduces load on the host CPUs, it may be beneficial for performance even if it does not improve network bandwidth or latency. A related possibility is to use a substitute for TCP/IP over a real network. This may improve performance if the device implementation can take advantage of specialized features of the real network, such as reliability and order guarantees provided by Fast Messages [61].

Our paravirtual file system prototype is faster than Linux NFS, and almost as fast as a conventional file system on a virtual disk (see Section 6.1.3). It also allows all the VMs that use it to share caches, reducing memory requirements.

Chapter 4

Extreme Paravirtualization

The previous chapter explained reasons to introduce extreme paravirtualization into a virtual machine environment. This chapter explains the idea of extreme paravirtualization in more detail, by describing extreme paravirtualization interfaces designed as network and file system modules, respectively. In the first section, we describe the requirements that these virtual interfaces make on the hosting virtual machine monitor. The following sections then describe our network and file system paravirtualization designs and prototypes in detail.

4.1 VMM Requirements

We assume the existence of a virtual machine monitor that runs directly on the hardware of a machine and that is simple enough to be trustworthy, that is, to have no bugs that subvert isolation among VMs. We also assume that the virtual machine monitor supports inter-VM communication (IVC) mechanisms, so that device drivers, etc. may run in separate virtual machines. Several research and industrial VMMs fall in this category, including Xen, Microsoft Viridian, and VMware ESX Server [48, 62, 63].

We require support for three IVC primitives: statically shared memory regions for implementing shared data structures such as ring buffers, the ability for one VM to temporarily grant access to some of its pages to a second VM, and a "doorbell" or *remote interrupt* mechanism for notifying a VM with a interrupt. These communication mechanisms are

supported by multiple modern VMMs, including VMware Workstation through its Virtual Machine Communication Interface [64], and related mechanisms have a history dating back to at least 1979 [65].

The attractions of shared memory include simplicity, ubiquity, and performance. Remote procedure call (RPC) is a viable alternative to shared memory, but RPC would have required putting more code into the VMM and possibly required communicating VMs to call into the VMM as an intermediary. It also would have forced more policy decisions into the VMM: should RPC calls be synchronous or asynchronous? what should be the form of and limits on arguments and return values? and so on. Finally, RPC can be effectively layered on top of shared memory, as we do in our POFS prototype 4.3.1.

An unfortunate pitfall of sharing memory between mutually distrusting parties is the possibility of data races: much like access to user-space data from a conventional kernel, data may change from one read to the next. Data that is writable by both parties is particularly troublesome, because a party cannot assume that data it writes one moment will not be maliciously overwritten by the other party in the next moment. It also increases total memory requirements, because the shared memory cannot safely store data of value to the party writing it, only copies of it.

Our IVC protocols reduce this risk of data races by using shared memory that is writable by one VM or the other, but never by both. For two-way communication, we use one set of pages that are accessible read/write by one VM and read-only by the other, and a second set of pages set up the opposite way.

We designed our paravirtual interfaces to be OS-neutral, in the hope that our service VMs could be useful with diverse application VMs, including Windows and Unix-like OSes other than Linux, not just those running the particular Linux version that we used. For example, the inode data structure in our file system interface does not correspond directly to Linux in-memory inode structure's layout or to an on-disk inode layout. Rather, we defined an independent format. Thus, it is necessary to do some data copying and format translation between them, which costs time and memory. However, it is also more robust: changes to OS data structures only require updating one piece of translation code, and in some cases this may happen automatically as a result of recompilation.

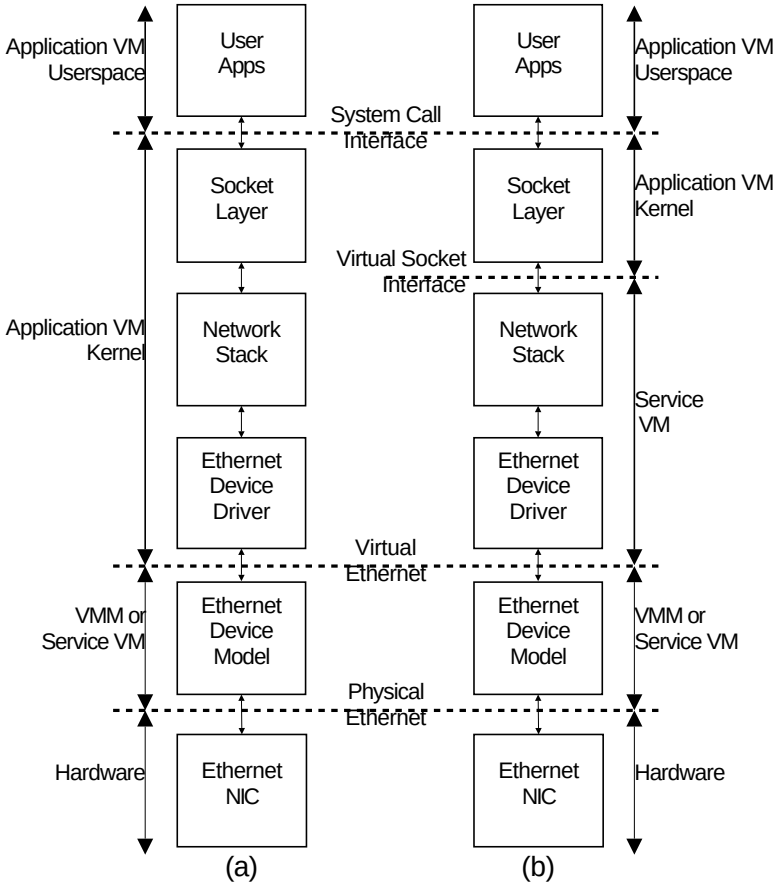

Figure 4.1: Networking architectures for virtual machine monitors: (a) the customary approach, and (b) extreme paravirtualization.

4.2 Network Paravirtualization

Modern operating system environments have evolved to implement networking protocols such as TCP/IP using multiple cleanly separated internal interfaces. Today's operating systems, including Linux, other Unix-like systems, and Windows [66], include at least the following interfaces in the networking stack:

- The system call interface accessed by user programs. Operations at this layer include the system calls `socket`, `bind`, `connect`, `listen`, `accept`, `send`, and `recv`.
- The virtual socket interface, which in effect translates between a system call-like

interface and TCP/IP or UDP/IP packets. This roughly corresponds to the transport and network layers.

- The virtual network device interface, which encapsulates packets inside frames and transmits them on a physical or virtual network. The essential operations at this layer are sending and receiving frames. It implements the data link and physical layers of the network.

Figure 4.1(a) shows the most common approach to networking for virtual machines. It is used by research and commercial VMMs from most vendors, including VMware, Microsoft, and Xen. In this approach, user applications use the system call interface to make network requests through the kernel's socket layer, which uses the kernel's network stack to drive a virtual Ethernet device. In a VMM that supports unmodified guest OSes, the virtual Ethernet device resembles a physical Ethernet device; in a paravirtual VMM, it has a streamlined interface. Regardless, the VMM or a privileged service VM in turn routes frames between the virtual Ethernet device and physical network hardware.

For this thesis we investigated paravirtualization at the virtual socket interface, as shown in Figure 4.1(b), with Linux as the application VM's guest operating system. We created a new Linux protocol family implementation that transparently substitutes for the native Linux TCP/IP stack. From user-level applications our network stack is accessed identically to conventional Linux TCP/IP stack. From an application's point of view, all system calls on sockets and file descriptors behave in the same way on these sockets as they do on sockets supplied by the Linux TCP/IP stack.

Instead of operating at a link level, using a packet interface, our paravirtualized network device, called *pull out networking* or *PON*, operates at the socket interface. It offers both TCP/IP-compatible reliable byte stream protocols and UDP-compatible datagram communication.

Each PON socket is represented by a data structure in shared memory. This socket data structure is divided into two pieces, one in shared memory that is writable by the VM and the other in shared memory writable only by the PON paravirtual network implementation. Each part includes connection state information and consumer/producer pointers into data buffers.

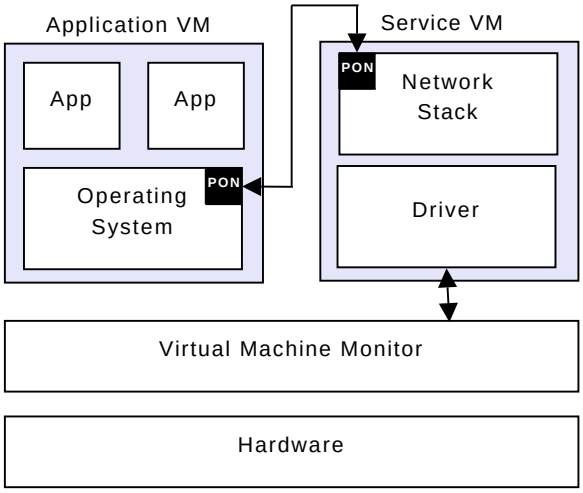

Figure 4.2: Accessing a physical network through a gateway VM using PON.

Implementing a TCP/IP stack in our VMM would violate the principle that the VMM should be simple enough that it can be assumed secure, so we instead placed the network stack in another VM and used the PON protocol to access it. We then made the network stack VM act as a gateway between the application VM and a conventional TCP/IP network, as shown in Figure 4.2. Thus, we effectively pulled the networking stack out of our application VM and moved it into a service VM.

4.2.1 Implementation Details

Multiple projects have layered network-like protocols on top of shared memory for communication between physical or virtual machines. Virtual memory-mapped communication (VMMC), for example, uses memory shared among multicomputer nodes as a basis for higher-level protocols, and Xenidc uses shared memory to implement high-level, network-like communication between device drivers running in VMs on a single host [67, 68]. PON adapts these ideas to transparently provide a substitute for real network protocols.

PON uses a ring buffer of pointers to sockets (actually, a pair of ring buffers, one modified by the application VM, the other by the paravirtualized network stack) to draw attention to sockets that require attention. A remote interrupt requests a look at the ring buffer.

To establish a TCP-like byte stream connection, the application VM initializes a new

socket structure, puts a pointer to it in the ring buffer, and sends a remote interrupt to the network stack. The PON network stack then initiates a TCP/IP connection on the external network, finishes initializing the socket, and replies with a pointer to the socket. (If the connection fails, it instead replies with an error code.)

To send data on a stream socket, the sender allocates a buffer from its shared memory and writes the data to it. It then indicates the buffer's location and size in the socket structure and adds a notification to the command queue. The paravirtual network stack updates a bytes received counter in the socket as data are acknowledged by the remote TCP endpoint.

To send more data on the socket, the application VM uses its existing buffer as a circular queue, inserting more data and pushing it to the receiver. Old data may be overwritten as the PON network stack indicates that it has been processed.

PON's buffer management technique, in which the data sender is responsible for managing its own buffer space, is a form of *sender-based buffer management* [69, 67, 61]. PON's implementation differs from some others in that buffer space is reserved at the time an application VM first accesses the paravirtual network device, instead of requiring an additional per-connection round trip. Because buffer space is drawn from shared memory owned by the sender, not by the receiver, there is no potential for a sender to cause a denial-of-service attack on its peer across a PON link through memory exhaustion.

UDP-like connectionless datagram messages are sent in a similar way. The application allocates shared memory and copies the message payload into it. Then it allocates a socket and initializes it with the message type, a pointer to the payload, and other metadata. Finally, it appends a pointer to the socket to the ring buffer and sends a remote interrupt. When the paravirtual network stack indicates that it has sent the message, the socket and its payload is freed up for other use. Further messages allocate new sockets.

We have not implemented rarely used TCP features, such as urgent data and simultaneous open, but they pose no fundamental difficulties.

4.3 File System Paravirtualization

Unix-like operating systems and Windows, among others, have long broken up their file system processing into the following multiple, cleanly separated layers (and sometimes

more) [70, 66]:

- File-related system calls by user programs request operations such as `open`, `creat`, `read`, `write`, `rename`, `mkdir`, and `unlink`.
- The virtual file system (VFS) layer calls into the individual file system's implementation of a system call. The file system executes it in a file system-specific way, in effect translating these operations into block read and write operations (for a disk-based file system).
- The block device layer performs read and write operations on device sectors.

In the most common approach to storage for virtual machines, all of these layers are implemented inside a single virtual machine's kernel. The VMM or a privileged service VM then implements a virtual (or paravirtual) disk device model.

However, we can use these internal interfaces to break pieces of a file system out of an operating system kernel at another layer. For this thesis, we investigated paravirtualization at the VFS layer, by implementing a new file system that acts, from the user's point of view, like any other Linux file system. Internally, instead of being layered on top of a conventional block device (or network), our file system transmits file system operations to a paravirtualized file system device, using a shared memory interface we call the *pull out file system* protocol or *POFS*. The paravirtualized file system device accessed through the POFS protocol can implement the file system in any way deemed appropriate: on top of a block device or a network, generated algorithmically, etc.

Compared to a network file system, POFS offers better performance (see section 6.1.3). Also, its semantics are closer to those of a local disk than those of most network file systems, in particular regarding cache coherence: data in a POFS file system is completely coherent for inter-VM access because data pages are physically shared between VMs.

POFS is particularly well-suited as a basis for implementing a *virtualization aware file system* (VAFS), a file system that combines the advantages of network file systems and virtual disk-based file systems. A VAFS provides the powerful versioning model and easy provisioning of virtual disks, while adding the fine-grained controlled sharing of distributed

file systems. The following chapter describes the idea of a VAFS, as well as our prototype implementation, in more detail.

The following section describes implementation details for our POFS prototype.

4.3.1 Implementation Details

The POFS interface uses a pair of small shared memory regions as ring buffers of RPC requests and replies. The application VM issues an RPC request to initiate an operation, to which the POFS implementation responds after the operation has completed. An RPC request/reply pair exists to implement most file system operations: `creat`, `read`, `write`, `rename`, `mkdir`, `unlink`, and so on. This use of a shared memory ring buffer for RPCs in a file system is adapted from VNFS [71].

File system operations, that work with regular file data, such as `read` and `write`, are handled differently, through RPCs that directly obtain access to data pages in the POFS server's file cache, with read/write access if the client VM is authorized to write to the file and read-only access otherwise. The application VM then performs the operation directly on the mapped memory. The data is cached in the application VM, so that later accesses do not incur the cost of an RPC. The guest physical frames can be further mapped as user virtual pages to implement the `mmap` system call.

This use of grant access to pages allows for cache consistency within a host with minimal overhead. When two client VMs access the same data or metadata, a single machine page is mapped into the "physical" memory of both, so that file modifications by one client are immediately seen by the others. This also reduces memory requirements.

Our implementation suffers from some races between file truncation (with `truncate`) and writes that extend files. The result is momentary cache incoherence between clients. We do not believe that this limitation is fundamental to our approach.

To improve the performance of access to file system metadata, e.g. for the `stat` or `access` system calls, we use a similar caching mechanism for metadata. The POFS interface exports POFS-format inodes for describing files. This inode format is largely file system and OS kernel independent. The POFS interface exports a cache of several pages of memory (20 pages, in our tests) that contain these POFS-format inodes packed into slots

(approximately 64 inodes per page). An application VM is granted read-only access to all of these cache pages, a single set of which are shared among all clients of a given POFS file system.

Using this cache, inode lookups can avoid RPC communication of the POFS interface. Each RPC that accesses an inode ensures that the corresponding POFS inode information is available and up-to-date in one of these cache slots and returns a pointer to its slot. This use of shared memory for inode data resembles VNFS [71].

To reduce memory usage, there are relatively few POFS inode cache slots, compared to the number of inodes cached by Linux at any given time, so clients must check that the inode in a slot is the one expected and, if it has been evicted, request that it be brought back into a cache slot. Our implementation chooses POFS inode cache slots for eviction randomly, which provides acceptable performance in our tests (see Section 6.1.3).

Our prototype implementation does not limit access to cached inodes only to VMs that can access those inodes. This is a security issue, since this can allow a VM to view attributes, such as modification times and link counts, of inodes that it otherwise would not be able to see. Inodes do not include file names, nor does the ability to view a cached inode give a VM any other ability to inspect or manipulate the inode or any of the files that reference it. Still, for security a more advanced implementation would group inodes into pages based on the credentials require to access them, and then allow VMs to view only the inodes that they are authorized to see.

POFS is robust against uncooperative or crashed client VMs. Each client is limited in the number of pages owned by the POFS server that it may map at any time. At the limit, to map a new page the client must also agree to unmap an old one. Client VMs may only map data and access metadata as file permissions allow.

Chapter 5

Virtualization Aware File Systems

The previous chapter described network and file system extreme paravirtualization prototypes in detail. This chapter further extends the extreme paravirtualization approach for virtual storage, by proposing the concept of a *virtualization aware file system* (VAFS) that combines the features of a virtual disk with those of a distributed file system. Whereas the features of an extreme paravirtualization file system, such as POFS, are comparable to a network file system, a VAFS adds versioning and other features that are particularly useful in a virtual environment.

5.1 Motivation

Virtual disks, the main form of storage in today's virtual machine environments, have many attractive properties, including a simple, powerful model for versioning, rollback, mobility, and isolation. Virtual disks also allow VMs to be created easily and stored economically, freeing users to configure large numbers of VMs. This enables a new usage model in which VMs are specialized for particular tasks.

Unfortunately, virtual disks have serious shortcomings. Their low-level isolation prevents shared access to storage, which hinders delegation of VM management, so users must administer their own growing collections of machines. Rollback and versioning takes place at the granularity of a whole virtual disk, which encourages mismanagement and reduces security. Finally, virtual disks' lack of structure obstructs searching or retrieving data in

their version histories [72].

Conversely, existing distributed file systems support fine-grained controlled sharing, but not the versioning, isolation, and encapsulation features that make virtual disks so useful.

To bridge the gap between these two worlds, we present Ventana, a *virtualization aware file system* (VAFS). Ventana extends a conventional distributed file system with versioning, access control, and disconnected operation features resembling those available from virtual disks. This obtains the benefits of virtual disks, without compromising usability, security, or ease of management.

Unlike traditional virtual disks whose allocation and composition is relatively static, in Ventana storage is ephemeral and highly composable, being allocated on demand as a *view* of the file system. This allows virtual machines to be rapidly created, specialized, and discarded, minimizing the storage and management overhead of setting up a new machine.

Virtual machines are changing the way that users perceive a "machine." Traditionally, machines were static entities. Users had one or a few, and each machine was treated as general-purpose. The design of virtual machines, and even their name, has largely been driven by this perception.

However, virtual machine usage is changing as users discover that a VM can be as temporary as a file. VMs can be created and destroyed at will, checkpointed and versioned, passed among users, and specialized for particular tasks. Virtual disks, that is, files used to simulate disks, aid these more dynamic uses by offering fully encapsulated storage, isolation, mobility, and other benefits that will be discussed fully in the following section.

Before that, to motivate our work, we will highlight the significant shortcomings of virtual disks. Most importantly, virtual disks offer no simple way to share read and write access between multiple parties, which frustrates delegating VM management. At the same time, the dynamic usage model for VMs causes them to proliferate, which introduces new security and management risks and makes such delegation sorely needed [73, 74].

Second, although it is easy to create multiple hierarchical versions of virtual disks, other important activities are difficult. A normal file system is easy to search with command-line or graphical tools, but searching through multiple versions of a virtual disk is a cumbersome, manual process. Deleting sensitive data from old versions of a virtual disk is similarly difficult.

Finally, a virtual disk has no externally visible structure, which forces entire disks to roll back at a time, despite the possible negative consequences [73]. Whether they realize it or not, whole-disk rollback is hardly ever what people actually want. For example, system security precludes rolling back password files, firewall rules, encryption keys, and binaries patched for security, and functionality may be impaired by rolling back network configuration files. Furthermore, the best choice of version retention policy varies from file to file [75], but virtual disks can only distinguish version policies on a whole-disk level.

These limitations of virtual disks led us to question why they are the standard form of storage in virtual environments. We concluded that their most compelling feature is compatibility. All of their other features can be realized in a network file system. By adopting a widely used network file system protocol, we can even achieve reasonable compatibility.

The following section details the virtual disk features that we wish to integrate into a network file system. The design issues raised in this integration are then covered in Section 5.3.

5.2 Virtual Disk Features

Virtual disks are, above all, backward compatible, because they provide the same block-level interface as physical disks. This section examines other important features that virtual disks offer, such as versioning, isolation, and encapsulation, and the usage models that they enable. This discussion shapes the design for Ventana presented in the next section.

5.2.1 Versioning

Because any saved version of a virtual machine can be resumed any number of times, VM histories take the form of a tree. Consider a user who “checkpoints” or “snapshots” a VM, permanently saving the current version as version 1. He uses the VM for a while longer, then checkpoints it again as version 2. So far, the version history is linear, as shown in Figure 5.1(a). Later, he again resumes from version 1, uses it for a while, then snapshots it another time as version 3. The tree of VMs now looks like Figure 5.1(b). The user can resume any version any number of times and create new snapshots based on these existing

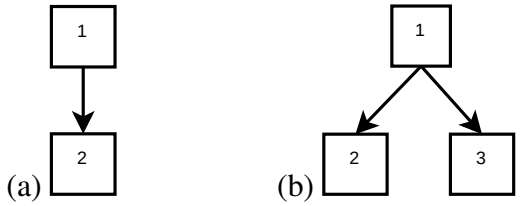

Figure 5.1: Snapshots of a VM: (a) first two snapshots; (b) after resuming again from snapshot 1, then taking a third snapshot.

versions, expanding the tree.

Virtual disks efficiently support this tree-shaped version model. A virtual disk starts with an initial or "base" version that contains all blocks (all-zero blocks may be omitted), corresponding to snapshot 1. The base version may have any number of "child" versions, and so may those versions recursively. Thus, like virtual machines, the versions of virtual disks form a tree. Each child version contains only a pointer to its parent and those blocks that differ from its parent. This copy-on-write sharing allows each child version to be stored in space proportional to the differences between it and its parent. Some implementations also support content-based sharing that shares identical blocks regardless of parent/child relationships.

Virtual disk versioning is useful for short-term recovery from mistakes, such as inadvertently deleting or corrupting files, or for long-term capture of milestones in configuration or development of a system. Linear history also effectively supports these usage models. But hierarchical versions offer additional benefits, described below.

Specialization

Virtual disks enable versions to be used for specialization, analogous to the use of inheritance in object-oriented languages. Starting from a base disk, one may fork multiple branches and install a different set of applications in each one for a specialized task, then branch these for different projects, and so on. This is easily supported by virtual disks, but today's file systems have no close analogue.

Non-Persistence

Virtual disks support "non-persistent storage." That is, they allow users to make temporary changes to disks during a given run of a virtual machine, then throw away those changes once the run is complete. This usage pattern is handy in many situations, such as software testing, education, electronic "kiosk" applications, and honeypots. Traditional file systems have no concept of non-persistence.

5.2.2 Isolation

Everything in a virtual machine, including virtual disks, exists in a protection domain decoupled from external constraints and enforcement mechanisms. This supports important changes in what users can do.

Orthogonal Privilege

With the contents of the virtual machine safely decoupled from the outside world, access controls are put into the hands of the VM owner (often a single user). There is thus no need to couple them to a broader notion of principals. Users of a VM are provided with their own "orthogonal privilege domain." This allows the user to use whatever operating systems or applications he wants, at his discretion, because he is not constrained by the normal access control model restricting who can install what applications.

Name Space Isolation

VMs can serve in the same role filled by `chroot`, BSD `jails`, application sandboxes, and similar mechanisms. An operating system inside a VM can even be easier to set up than more specialized, OS-specific jails that require special configuration. It is also easier to reason about the security of such a VM than about specialized OS mechanisms. A key reason for this is that VMs afford a simple mechanism for name space isolation, i.e. for preventing an application confined to a VM modifying outside system resources. The VM has no way to name anything outside the VM system without additional privilege, e.g. access to a shared network. A secure VMM can isolate its VMs perfectly.

5.2.3 Encapsulation

A virtual disk fully encapsulates storage state. Entire virtual disks, and accompanying virtual machine state, can easily be copied across a network or onto portable media, notebook computers, etc.

Capturing Dependencies

The versioning model of virtual disks is coarse-grained, at the level of an entire disk. This has the benefit of capturing all possible dependencies with no extra effort from the user. Thus, short-term "undo" using a virtual disk can reliably back out operations with complex dependencies, such as installation or removal of a major application or device driver, or a complex, automated configuration change.

Full capture of dependencies also helps in saving milestones in the configuration of a system. The snapshot will not be broken by subsequent changes in other parts of the system, such as the kernel or libraries, because those dependencies are part of the snapshot [76].

Finally, integrating dependencies simplifies and speeds branching. To start work on a new version of a project or try out a new configuration, all the required pieces come along automatically. There is no need to again set up libraries or configure a machine.

Mobility

A virtual disk can be copied from one medium to another without retaining any tie to its original location. Thus, it can be used while disconnected from the network. Virtual disks thereby offer mobility, the ability to pick up a machine and go.

Merging and handling of conflicts has long been an important problem for file systems that support disconnected operation [77], but there is no automatic means to merge virtual disks. Nevertheless, virtual disks are useful for mobility, indicating that merging is not important in the common case. (In practice, when merging is important, users tend to use revision control systems.)

5.3 Virtualization Aware File System Design

This section describes Ventana, an architecture for a virtualization aware file system. Ventana resembles a conventional distributed file system in that it provides centralized storage for a collection of file trees, allowing transparency and collaborative sharing among users. Ventana's distinction is its versioning, isolation, and encapsulation features to support virtualization, based on virtual disk support for these same features,

The high-level architecture of Ventana can apply to various low-level architectures: centralized or decentralized, block-structured or object-structured, etc. We restrict this section to essential, high-level design elements. The following section discusses specific choices made in our prototype.

Ventana offers the following abstractions:

Branches Ventana supports VM-style versioning with *branches*. A *private branch* is created for use primarily by a single VM, making the branch effectively private, like a virtual disk. A *shared branch* is intended for use by multiple VMs. In a shared branch, changes made from one VM are visible to the others, so these branches can be used for sharing files, like a conventional network file system.

Non-persistent branches, whose contents do not survive across reboots are also provided, as are *volatile branches*, whose contents are never stored on a central server, and are deleted upon migration. These features are especially useful for providing storage for caches and cryptographic material that for efficiency or security reasons, respectively, should not be stored or migrated.

Branches are detailed in Section 5.3.1.

Views Ventana is organized as a collection of file trees. To instantiate a VM, a *view* is constructed by mapping one or more of these trees into a new file system name space. For example, a base operating system, add-on applications, and user home directories might each be mounted from a separate file tree.

This provides a basic model for supporting name space isolation and allows for rapid synthesis of new virtual machines, without the space or managment overhead normally associated with setting up a new virtual disk.

Section 5.3.2 describes views in more detail.

Access Control File permissions in Ventana must satisfy two kinds of needs: those of the guest OSes to partition functionality according to the guests' own principals, and those of users to control access to confidential information. Ventana provides two types of *file ACLs* that satisfy these two kinds of orthogonal needs.

Ventana also offers *branch ACLs* which support common VM usage patterns, such as one user granting others permission to clone a branch and modify the copy (but not the original), and *version ACLs* which alleviate security problems introduced by file versioning.

Section 5.3.3 describes access control in Ventana.

Disconnected Operation Ventana allows for a very simple model of mobility by supporting disconnected operation, through a combination of aggressive caching and versioning. Section 5.3.4 talks about disconnected operation in Ventana.

5.3.1 Branches

Some conventional file systems support versioning of files and directories. Details vary regarding which versions are retained, when older versions are deleted, and how older versions are named. However, in all of them, versioning is "linear," that is, at any point in each file has a unique latest version.

When versions form a tree that grows in more than one direction, the "latest version" of a file can be ambiguous. The file system must provide a way for users to express where in the tree to look for a file version.

To appreciate these potential ambiguities, consider an example. Ziggy creates a VM and allows Yves, Xena, and Walt to each fork it a personalized version. The version tree for a file personalized by each person would look something like Figure 5.2(a). If an access to a file by default refers to the latest version anywhere in the tree, then each person's changes would appear in the others' VMs. Thus, the tree of versions would act like a chain of linear versions.

In a different situation, suppose Vince and Uma use a shared area in the file system for collaboration. Most of the time, they do want to see the latest version of a file. Thus,

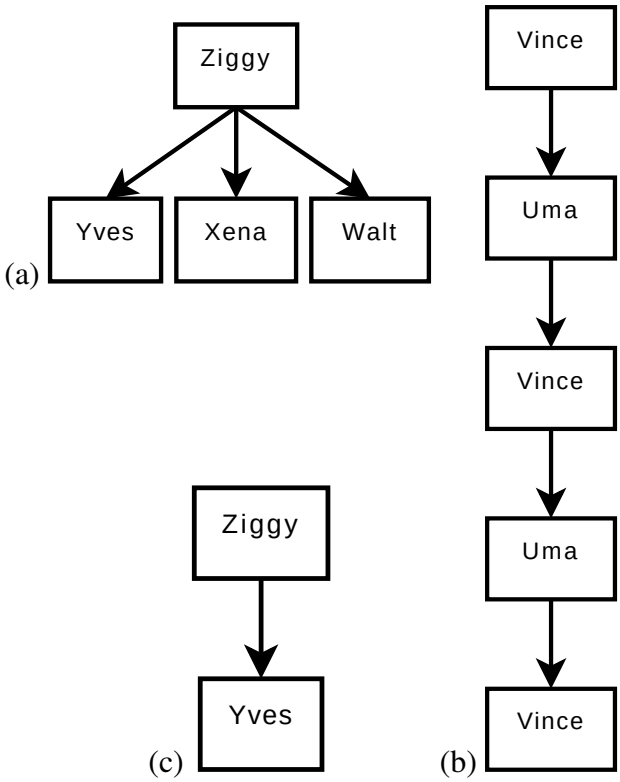

Figure 5.2: Trees of file versions when (a) Ziggy allows Yves, Xena, and Walt to fork personalized versions of his VM; (b) Vince and Uma collaboratively edit a file; and (c) Ziggy's VM has been forked by Yves, as in (a), but not yet by Xena or Walt.

the version history of such a file should be linear, with each update following up on the previous one, resembling Figure 5.2(b).

The essential difference between these two cases is intention. The version tree alone cannot distinguish between desires for shared or personalized versions of the file system without knowledge of intention.

Consider another file in Ziggy's VM. If only Yves has created a personalized version of the file, then the version tree looks like Figure 5.2(c). The shape of this tree cannot be distinguished from an early version of Figure 5.2(a) or (b). Thus, Ventana must provide a way for users to specify their intentions.

Private and Shared Branches

Ventana introduces *branches* to resolve version ambiguity. A branch is a linear chain in the tree of versions. Because a branch is linear, the latest version or the version at a particular time is unambiguous for a given file in a specified branch.

A branch begins as an exact copy of the contents of some other branch at the current time, or at a chosen earlier time. After creation, the new branch and the branch that was copied are independent, so that modifying one has no effect on the other.

Branches are created by copying. Thus, multiple branches may contain the same version of a file. Therefore, for a file access to be unambiguous, both a branch and a file must be specified. Mounting a tree in a virtualization aware file system requires specifying the branch to mount.

If a single client wants a private copy of the file tree, a *private branch* is created for its exclusive use. Like a file system on a virtual disk, a private branch will only be modified by a single client in a single VM, but in other respects it resembles a conventional network file system. In particular, access to files by entities other than the guest that "owns" the branch is easily possible, enabling centralized management such as scanning for malware, file backup, and tracking VM version histories.

If multiple clients mount the same branch of a Ventana file tree, then those clients see a shared view of the files it contains. As in a conventional network file system, a change made by one client in such a *shared branch* will be immediately visible to the others. Of course, propagation of changes between clients is still subject to the ordinary issues of cache consistency in a network file system.

The distinction between shared and private branches is simply the number of clients expected to write to the branch. If necessary, centralized management tools can modify files in a so-called "private" branch (e.g. to quarantine malware) but this is intended to be uncommon. Either type of branch might have any number of read-only clients.

A single file might have versions in shared and private branches. For example, a shared branch used for collaboration between several users might be forked off into a private branch by another user for some experimental changes. Later, the private branch could be discarded or consolidated into the shared branch.

Other Types of Branches

In addition to shared and private branches, there are several other useful qualifiers to attach to file trees.

Files in a *non-persistent branch* are deleted when a VM is rebooted. These are useful for directories of temporary files such as `/tmp`.

Files in a *volatile branch* are also deleted on reboot. They are never stored permanently on the central server, and are deleted when a VM is migrated from one physical machine to another. They are useful for caches (e.g. `/var/cache` on GNU/Linux) that need not be migrated and for storing security tokens (e.g. Kerberos tickets) that should not reside on a central server.

Maintaining any version history for some files is an inherent security risk [73]. For example, the OpenSSL cryptography library stores a "random seed" file in the file system. If this is stored in a snapshot, every time a given snapshot is resumed, the same random seed will be used. In the worst case, we will see the same sequence of random numbers on every execution. Even in the best case, its behavior may be easier to predict, and if old versions are kept, then it may be possible to guess past behavior (e.g. keys generated in past runs).

Ventana offers *unversioned files* as a solution. Unversioned files are never versioned, whether linearly or in a tree. Changes always evolve monotonically forward with time. Applications for unversioned files include storing cryptographic material, firewall rules, password files, or any other configuration state where rollback would be problematic.

5.3.2 Views

Ventana is organized as a set of file trees, each of which contains related files. For example, some file trees might contain root file systems for booting various operating systems (Linux, Windows XP, ...) and their variants (Debian, Red Hat, SP1, SP2, ...). Another might contain file systems for running various local or specialized applications. A third would have a hierarchy for each user's files.

Creating a new VM mainly requires synthesizing a *view* of the file system for the VM. This is accomplished by mapping one or more trees (or parts of trees) into a new name

space. For example, the Debian root file system might be combined with a set of applications and user home directories. Thus, OSes, applications, and users can easily "mix and match" in a Ventana environment.

Whether each file tree in a view is mounted in a shared or a private branch depends on the user's intentions. The root file system and applications could be mounted in private branches to allow the user to update and modify his own system configuration. Alternatively, they could be mounted in shared branches (probably read-only) to allow maintenance to be done by a third party. In the latter case, some parts of the file system would still need to be private, e.g. `/var` under GNU/Linux. Home directories would likely be shared, to allow the user to see a consistent view of his and others' files regardless of the VM viewing them.

5.3.3 Access Control

Access control is different in virtual disks and network file systems. On a virtual disk, the guest OS controls every byte. The guest OS is responsible for tracking ownership and permissions and making access control decisions in the file system. The virtual disk itself has no access control responsibility. A VAFS cannot use this scheme, because allowing every guest OS to access any file, even those that belong to other VMs, is obviously unacceptable. At a minimum, there must be enough control in the system to prevent abuse.

Access control in a conventional network file system is the reverse of the situation for a virtual disk. The file server is ultimately in charge of access control. As a network file system client, a guest OS can deny access to its own processes, but it cannot override the server's refusal to grant access. Commonly, NFS servers deny access as the superuser ("squash root") and CIFS and AFS servers grant access only via principals authenticated to the network.

This style of access control is also, by itself, inappropriate in a VAFS. Ventana should not deny a guest OS control over its own binaries, libraries, and applications. If these were, for example, stored on an NFS server configured to "squash root," the guest OS would not be able to create or access any files as the superuser. If they were stored on a CIFS or AFS server, the guest OS would only be able to store files as users authenticated to the network.

In practice this would prevent the guest from dividing up ownership of files based on their function (system binaries, print server, web server, mail server, ...), as many systems do.

Ventana solves the problem of access control through multiple types of ACLs: *file ACLs*, *version ACLs*, and *branch ACLs*. For any access to be allowed, it must be permitted by all three applicable ACLs. Each kind of ACL serves a different primary purpose. The three types are described individually below.

File ACLs

File ACLs provide protection on files and directories that users conventionally expect and OSes conventionally provide. Ventana supports two types of file ACLs that provide orthogonal privileges. *Guest file ACLs* are primarily for guest OS use. Guest OSes have the same level of control over guest file ACLs that they do over permissions in a virtual disk. In contrast, *server file ACLs* provide protection that guest OSes cannot bypass, similar to permissions enforced by a conventional network file server.

Both types of file ACLs apply to individual files. They are versioned in the same way as other file metadata. Thus, revising a file ACL creates a new version of the file with the new file ACL. The old version of the file continues to have the old file ACL.

Guest file ACLs are managed and enforced by the guest OS using its own rules and principals. Ventana merely provides storage. These ACLs are expressed in the guest OS's preferred form. We have so far implemented only the 9-bit `rwxrwxrwx` access control lists used by the Unix-like guest OSes. Guest file ACLs allow the guest OS to divide up file privileges based on roles.

Server file ACLs, the other type of file ACL, are managed and enforced by Ventana and stored in Ventana's own format. Server file ACLs allow users to control access to files across all file system clients.

Version ACLs

A version ACL applies to a version of a file. They are stored as part of a version, not as file metadata, so that changing a version ACL does not create a new file version. Every version of a file has an independent version ACL. Conversely, when multiple branches contain the

same version of a file, that single version ACL applies in each case. Version ACLs are not versioned themselves. Like server file ACLs, version ACLs are enforced by Ventana itself.

Version ACLs are Ventana's solution to a class of security problem common to all versioning file systems. Suppose Terry creates a file and writes confidential data to it. Soon afterward, Terry realizes that the file's permissions incorrectly allow Sally to read it, so he corrects the permissions. In a file system without versioning, the file would then be safe from Sally, as long as she had not already read it. If the permissions on older file versions are fixed, however, Sally can still access the older version of the file.

A partial solution to Terry's problem is to grant access to older versions based on the current version's permissions, as Network Appliance filers do [78]. Now, suppose Terry edits a file to remove confidential information, then grants read permission to Sally. Under this rule, Sally can then view the older, confidential versions of the file, so this rule is also flawed.

Another idea is to add a permission bit to each file's metadata that determines whether a user may read a file once it has been superseded by a newer version, as in the S4 self-securing storage system [79]. Unfortunately, modifying permissions creates a new version (as does any change to file metadata) and only the new version is changed. Thus, this permission bit is effective only if the user sets it before writing confidential data, so it would not protect Terry.

Only two version rights exist. The "r" (read) version right is Ventana's solution to Terry's problem. At any time, Terry can revoke the read right on old versions of files he has created, preventing access to those file versions. The "c" (change) right is required to change a version ACL. It is implicitly held by the creator of a version. (Any given file version is immutable, so there is no "write" right.)

Branch ACLs

A branch ACL applies to all of the files in a particular branch and controls access to current and older versions of files. Like version ACLs, branch ACLs are accessed with special tools and enforced by Ventana.

The "n" (newest) branch right permits read access to the latest version of files in a branch. It also controls forking the latest version of the branch.

In addition to "n", the "w" (write) right is required to modify any files within a branch. A user who has "n" but not "w" may fork the branch. Then, as owner of the new branch, he may change its ACL and modify the files in the new branch. This does not introduce a security hole because the user may only modify the files in the new branch, not those in the old branch. The user's access to files in the new branch are, of course, still subject to Ventana file ACLs and version ACLs.

The "o" (old) right is required to access old versions of files within a branch. This right offers an alternative solution to Terry's problem of insecure access to old versions. If Terry controls the branch in which the old versions were created, then he can use its branch ACL to prevent other users from accessing old versions of any file in the branch. This is thus a simpler but less focused approach than adjusting the appropriate version ACL.

The "c" (change) right is required to change a branch ACL. It is implicitly held by the owner of a branch.

5.3.4 Disconnected Operation

Virtual disks can be used while disconnected from the network, as long as the entire disk has been copied onto the disconnected machine. Thus, for a virtualization aware file system to be as widely useful as a virtual disk, it must also gracefully tolerate network disconnection.

Research in network file systems has identified a number of features required for successful disconnected operation [77, 80, 81]. Many of these features apply to Ventana in the same way as conventional network file systems. Ventana, for example, can cache file system data and metadata on disk, which allows it to store enough data and metadata to last the period of disconnection. Our prototype caches entire files, not individual blocks, to avoid the need to allow reading only the cached part of a file during disconnection, which at best would be surprising behavior. Ventana can also buffer changes to files and directories and write them back upon reconnection. Some details of these features of Ventana are included in the description of our prototype (see Section 5.4).

Handling conflicts, that is, different changes to the same files, is a thorny issue in a design for disconnected operation. Fortunately, earlier studies of disconnection have shown conflicts to be rare in practice [77]. In Ventana conflicts may be even rarer, because they

cannot occur in private branches. Therefore, Ventana does not try to intelligently handle conflicts. Instead, changes by disconnected clients are committed at the time of reconnection, regardless of whether those files have been changed in the meantime by other clients, and announces what it is doing to the user. If manual merging is needed in shared branches, it is still possible based on old versions of the files. To make it easy to identify file versions just before reconnection, Ventana creates a new branch just before it commits the disconnected changes.

5.4 Implementation Details

To show that our ideas can be realized in a practical and efficient way, we developed a simple prototype of Ventana. This section describes the prototype's design and use.

The Ventana prototype is written in C. We developed it under Debian GNU/Linux "unstable" on 80*x*86 PCs running Linux 2.6.*x*, using VMware Workstation 5.0 as VMM. The servers in the prototype run as Linux user processes and communicate over TCP using the GNU C library implementation of ONC RPC [82].

Figure 5.3 outlines Ventana's structure, which is described in more detail below.

5.4.1 Server Architecture

A conventional file system operates on what Unix calls a block device, that is, an array of numbered blocks. Our prototype is instead layered on top of an *object store* [83, 84]. An object store contains *objects*, sparse arrays of bytes numbered from zero to infinity, similar to files. In the Ventana prototype, objects are immutable.

The object store consists of one or more *object servers*, each of which stores some of the file system's objects and provides a network interface for storing new objects and retrieving the contents of old ones. Objects are identified by randomly selected 128-bit integers called *object numbers*. Object numbers are generated randomly to allow them to be chosen without coordination between hosts. Collisions are unlikely as long as significantly fewer than 2^{64} have been generated, according to the "birthday paradox" [85]. Ventana does not attempt to detect collisions.

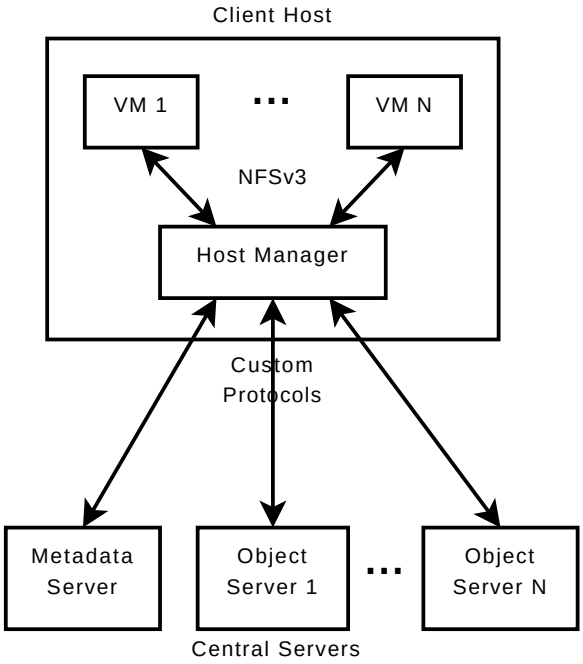

Figure 5.3: Structure of Ventana. Each machine whose VMs use Ventana runs a host manager. The host manager talks to the VMs over NFSv3 and to Ventana's centralized metadata and object servers over a custom protocol.

Each version of a file's data or metadata is stored as an object. When a file's data or metadata is changed, the new version is stored as a new object under a new object number. The old object is not changed and it may still be accessed under its original object number. However, this does not mean that every intermediate change takes up space in the object store, because client hosts (that is, machines that run Ventana clients in VMs) consolidate changes before they commit a new object.

As in an ordinary file system, each file is identified by an inode number, which is again a 128-bit, randomly selected integer. Each file may have many versions across many branches. When a client host needs to know what object stores the latest version of a file in a particular branch, it consults the *version database* by contacting the *metadata server*. The metadata server maintains the version database that tracks the versions of each file, the *branch database* that tracks the file system's branch structure, the database that associates branch names and numbers, and the database that stores VM configurations.

Scalability

The metadata server's databases are implemented using Berkeley DB, an embedded database engine that supports single-master replication across an arbitrary number of hosts. Single-master replication should allow Ventana to scale to the point where write requests overwhelm the master server. Because most metadata server RPC requests are read-only, overwhelming a master requires a large number of clients. Moreover, only writes to a shared branch have any urgent need to be committed to the metadata server, so other writes may be delayed if the metadata server is busy. Client-side caching also reduces load on the metadata and object servers.

Objects can be distributed among any number of object servers. The object server used to store a object is selected based on a hash of the object number, which tends to evenly distribute objects across the available object servers.

Availability

If a Berkeley DB master server fails, the remaining metadata servers may elect a new master using the algorithm built into Berkeley DB. If a network partition occurs with at least $n/2 + 1$ out of n metadata servers in one partition, then that partition, similarly, can elect a new master if necessary. Upon recovery in either case, the metadata servers automatically synchronize.

Object servers may also be replicated for availability. A hash of the object number can be used to select the object servers on which to store the object. If each object is stored on m object servers, then Ventana can tolerate loss of $m - 1$ or fewer object servers without data loss. Because objects are immutable, there is no need for protocols that ensure consistency between copies of an object.

5.4.2 Client Architecture

The *host manager* is the client-side part of the Ventana prototype. One copy of the host manager runs on each platform and services any number of local client VMs. Our prototype does not encapsulate the host manager itself in a VM.

For compatibility with existing clients, the host manager includes a NFSv3 [86] server for clients to use for file access. NFSv3 is both easy to implement and widely supported. Thus, any client operating system that supports NFSv3 can use a Ventana file system, including most Unix-like operating systems and Windows (with Microsoft's free Services for Unix).

The host manager maintains in-memory and on-disk caches of file system data and metadata. Objects may be cached indefinitely because they are immutable. Objects are cached in their entirety to simplify implementing the prototype and to enable disconnected operation (see Section 5.4.2). Records in the version and branch databases are also immutable, except for the ACLs they include, which change rarely. In a shared branch, records added to the version database to announce a new file version are a cache consistency issue, so the host manager checks the version database for new versions on each access (except when disconnected). In a private branch, normally only one client modifies the branch at a time, so that client's host manager can cache data in the branch for a long time (or until the client VM is migrated to another host), although other hosts should check for updates more often.

The host manager also buffers file writes. When a client writes a file, the host manager writes the modified file to the local disk. Further changes to the file are also written to the same file. If the client requests that writes be committed to stable storage, e.g. to allow the guest to flush its buffer cache or to honor an `fsync` call, then the host manager commits the modified files to the local disk. Commitment does not perform a round trip on a physical network.

Branch Snapshots

After some amount of time, the host manager takes a snapshot of outstanding changes within a branch. Users can also explicitly create (and optionally name) branch snapshots. A snapshot of a branch is created simply by forking the branch. Forking a branch copies its content, so this has the desired effect. In fact, copying occurs on a copy-on-write basis, so that the first write to any of the files in the snapshot creates and modifies a new copy of the file. Creating a branch also inserts a record in the branch database.

After it takes a snapshot, the host manager uploads the objects it contains into the object

store. Then, it sends records for the new file versions to a metadata server, which commits them to the version database in a single atomic transaction. The changes are now visible to other clients.

The host manager assumes that private branch data is relatively uninteresting to clients on other hosts, so it takes snapshots in private branches relatively rarely (every 5 minutes). On the other hand, other users may be actively using files in shared branches, so the host manager takes snapshots often (every 3 seconds).

Because branch snapshots are actually branches themselves, older versions of files can be viewed using regular file commands by first adding the snapshot branch to the view in use. Branches created as snapshots are by default read-only, to reduce the chance of later confusion if a file's "older version" actually turns out to have been modified.

Views and VMs

Multiple branches can be composed into a view. Ventana describes a view with a simple text format that resembles a Unix `fstab`, e.g.:

```
debian:/         /        shared,ro
home-dirs:/      /home    shared
bob-version:/    /proj    private
```

Each line describes a mapping between a branch, or a subset of a branch, and a directory within the view. We say that each branch is *attached* to its directory in the view.[1]

A VM is a view plus configuration parameters for networking, system boot, and so on. A VM could be described by the view above followed by these additional options:

```
-pxe-kernel debian:/boot/vmlinuz
-ram 64
```

Ventana provides a utility to start a VM based on such a specification. Given the above VM specification, it would set up a network boot environment (using the PXE protocol) to boot the kernel in `/boot/vmlinuz` in the `debian` branch, then launch VMware Workstation for the user to allow the user to interact with the VM.

[1] We use "attach" instead of "mount" because mounts are implemented inside an OS, whereas the guest OS that uses Ventana does not implement and is not aware of the view's composition.

VM Snapshots Ventana supports snapshots of VMs in just the same way as it supports snapshots of branches.[2] A snapshot of a VM is a snapshot of each branch in the VM's view combined with a snapshot of the VM's runtime state (RAM, device state, ...). To create a snapshot, Ventana snapshots the branches included in the VM, copies the runtime state file written by Workstation into Ventana as an unnamed file, and saves a description of the view and a pointer to the suspend file.

Later, another Ventana utility may be used to resume from the snapshot. When a VM snapshot is resumed, private branches have the contents that they did when the snapshot was taken, and shared branches are up-to-date. Ventana also allows resuming with a "frozen" copy of shared branches as of the time of the snapshot. Snapshots can be resumed any number of times, so resuming forks each private branch in the VM for repeatability.

Disconnected Operation

The host manager supports disconnected operation, that is, file access is allowed even without connectivity to the metadata and object server. Of course, access is degraded during disconnection: only cached files may be read, and changes in shared branches by clients on the other hosts are not visible. Write access is unimpeded. Disconnected operation is implemented in the host manager, not in clients, so all clients support disconnected operation.

We designed the prototype with disconnected operation in mind. Caching eliminates the need to consult the metadata and object servers for most operations, and on-disk caching allows for a large enough cache to be useful for extended disconnection. Whole-object caching avoids surprising semantics that would allow only part of a file to be read. Write buffering allows writing back changes to be delayed until reconnection.

We have not implemented user-configurable "hoarding" policies in the prototype. Implementing them as described by Kistler et al. [77] would be a logical extension.

Fixing NFS Warts

We used NFSv3 [86] as Ventana's file system access protocol because it is widely supported and because it is relatively easy to implement. However, it has a few warts that are difficult

[2] VMware Workstation has its own snapshot capability. Ventana's snapshot mechanism demonstrates VM snapshots might be integrated into a VAFS.

to avoid in a conventional file system design. This section describes how we designed around these problems in Ventana.

As discussed in Section 6.2.3, a more advanced implementation would, for performance, want to implement a protocol faster than NFSv3. However, any such protocol will require support to be added to guest OSes, so even such an implementation would want to support NFSv3 (or even NFSv2) for backward compatibility, in which case these notes would still be relevant.

Directory Reads The NFSv3 `READDIR` RPC read a group of directory entries. Each entry returned includes, among other fields, a file name and a "cookie" that the client can pass to a later call to indicate where to start reading. Most servers encode each cookie as a byte offset from the beginning of the directory. The `READDIR` response also includes a "cookie verifier" that the client passes back in later calls. The cookie verifier allows the server to return a "bad cookie" error if the directory changes between two `READDIR` calls. The NFSv3 specification suggests using the directory's modification time as the cookie verifier.

Unfortunately, NFS clients do not gracefully handle bad cookies. The Linux NFSv3 client, for example, passes the error to user programs, many of which give up on reading the rest of the directory. Servers should therefore report bad cookies rarely if ever, instead recovering from them as best they can. Usually this amounts to rounding the cookie to the nearest start of a directory entry, but this approach can return the same name twice within a directory, or omit names.

We designed the prototype's directory format to avoid the problem. Each directory entry includes a file name and an inode number, as in a traditional Unix file system, plus a "sequence number" that identifies when it was added. Each entry added to a given directory receives the next larger sequence number.

In a directory, then, cookies and cookie verifiers are sequence numbers. An initial `READDIR` returns the current maximum sequence number as the cookie verifier. Later calls skip over entries whose sequence numbers are greater than the cookie verifier. Thus, entries added after the first `READDIR` are not returned to the client. No duplicates will be returned, and no entries will be omitted. Calls to `READDIR` that restart reading the

```
                          ubuntu:/                 /                 shared,ro
                        home-dirs:/                /home             shared
                             none                  /tmp              non-persistent
12ff2fd27656c7c7e07c5ea1e2da367f:/var              /var              private
                         cad-soft:/                /opt/cad-soft     shared,ro
                           common:/etc/resolv.conf /etc/resolv.conf  shared,ro
                           common:/etc/passwd      /etc/passwd       shared,ro
8368e293a23163f6d2b2c27aad2b6640:/etc/hostname     /etc/hostname     private
b6236341bd1014777c1a54a8d2d03f7c:/etc/ssh/host_key /etc/ssh/host_key unversioned
```

Figure 5.4: Partial specification of the view for Bob's basic VM.

directory from the beginning receive a new cookie verifier, so new entries are not invisible to clients forever.

Deleted Files In a Unix file system, a file persists as long as any process has it open, regardless of whether all of the names for the file have been deleted. NFSv3, however, has no notion of an "open" file, so that when the last name for a file is deleted, the file is gone, and its file handle becomes invalid. Some NFSv3 clients, to avoid the problem, rename an open file to a new hidden name instead of removing it, as suggested by the NFSv3 specification.

Ventana cannot prevent clients from attempting to work around this issue. However, it does not require the workaround: files in Ventana persist beyond the deletion of their final link, in the form of older versions. An NFSv3 file handle for a deleted file in Ventana continues to function in the same way as a file handle for an existing file.

5.5 Usage Scenario

This section presents a scenario for use of Ventana and shows how, in this setting, Ventana offers a better solution than both virtual disks and network file systems.

5.5.1 Scenario

We set our scene at Widgamatic, a manufacturer and distributor of widgets.

```
                    carl-debian:/                 /                 private
                      home-dirs:/                 /home             shared
                           none                   /tmp              non-persistent
                         common:/etc/resolv.conf  /etc/resolv.conf  shared,ro
                         common:/etc/passwd       /etc/passwd       shared,ro
b6236341bd1014777c1a54a8d2d03f7c:/etc/ssh/host_key /etc/ssh/host_key unversioned
```

Figure 5.5: Partial specification of the view for Carl's custom VM.

Alice the Administrator

Alice is Widgamatic's system administrator in charge of virtual machines. Software used at Widgamatic has diverse requirements, and Widgamatic's employees have widely varying preferences. Alice wants to accommodate everyone as much as she can, so she supports various operating systems: Debian, Ubuntu, Red Hat, and SUSE distributions of GNU/Linux, plus Windows XP and Windows Server 2003. For each of these, Alice creates a shared branch and installs the base OS and some commonly used applications. She sets the branch ACLs to allow any user to read, but not write, these branches.

Alice creates `common`, a second shared branch, to hold files that should be uniform across the company, such as `/etc/hosts` and `/etc/resolv.conf`. Again, she sets branch ACLs to grant other users read-only access.

Alice also creates a shared branch for user home directories, called `home-dirs`, and adds a directory for each Widgamatic user in the root of this branch. Alice sets the branch ACL to allow any user to read or write the branch, and server file ACLs so that, by default, each user can read or write only his (or her) home directory. Users can of course modify server file ACLs in their home directories as needed.

Bob's Basic VM

Bob is a Widgamatic user with basic needs. Bob uses a utility written by Alice to create a Linux-based VM primarily from shared branches. Figure 5.4 shows part of the specification written by this utility.

The root of Bob's VM is attached to the Ubuntu shared branch created by Alice. This branch's ACL prevents Bob modifying files in the branch (it is attached read-only besides). The Linux file system is well suited for this situation, because its top-level hierarchies

segregate files based on whether they can be attached read-only during normal system operation. The `/usr` tree is an example of a hierarchy that normally need not be modifiable.

The `/home` and `/tmp` trees are the most prominent examples of hierarchies that must be writable, so Bob's VM attaches a writable shared branch and a non-persistent branch, respectively, at these points. Keyword `none` in place of a branch name in `/tmp`'s entry causes an initially empty branch to be attached.

The `/var` hierarchy must be writable and persistent, and it cannot be shared between machines. Thus, Alice's utility handles `/var` by creating a fork of the Ubuntu branch, then attaching the forked branch's `/var` privately in the VM. The utility does not give the forked branch a name, so the VM specification gives the 128-bit branch identifier as 32 hexadecimal digits.

Bob needs to use the company's CAD software to design widgets, so the CAD software distribution is attached into his VM.

Most of the VM's configuration files in `/etc` receive their contents from the Ubuntu branch attached at the VM's root. Some, e.g. `/etc/resolv.conf` and `/etc/passwd` shown here, are attached from Alice's "common files" branch. This allows Alice to update a file in just that branch and have the changes automatically reflected in every VM. A few, such as `/etc/hostname` shown here, are attached from private branches to allow their contents to be customized for the particular VM. Finally, data that should not be versioned at all, such as the private host key used to identify an SSH server, is attached from an unversioned branch. The latter two branches are, like the `/var` branch, unnamed.

Bob's VM, and VMs created in similar ways, would automatically receive the benefits of changes and updates made by Alice as soon as she made them. They would also see changes made by other users to their home directories as soon as they occur.

Carl's Custom VM

Carl wants more control over his VM. He prefers Debian, which is available as a branch maintained by Alice, so he can base his VM upon Alice's. Carl forks a private branch from Alice's Debian branch and names the new branch `carl-debian`.

Carl integrates his branch into a VM of his own, using a specification that in part looks like Figure 5.5. Carl could write this specification by hand, or he might choose to start from

one, like Bob's, generated by Alice's utility. Using a private branch as root directory means that Carl need not attach private branches on top of `/var` or `/etc/hostname`, making Carl's specification shorter than Bob's.

Even though Carl's base operating system is private, Carl's VM still attaches many of the same shared branches that Bob's VM does. Shared home directories and common configuration files ease Carl's administrative burden just as they do Bob's. He could choose to keep private copies of these files, but to little obvious benefit.

Carl must do more of the work of administering his own system, because Alice's changes to shared branches do not automatically propagate to his private branch. Carl could use Ventana to observe how the parent `debian` branch has changed since the fork, or Alice could monitor forked branches to ensure that important patches are applied in a timely fashion.

Alice in Action

One morning Alice reads a bulletin announcing a critical security vulnerability in Mozilla Firefox. Alice must do her best to make sure that the vulnerable version is properly patched in every VM. In a VM environment based on virtual disks, this would be a daunting task. Ventana, however, reduces the magnitude of the problem considerably.

First, Alice patches the branches that she maintains. This immediately fixes VMs that use her shared branches, such as Bob's VM.

Second, Alice can take steps to fix others' VMs as well. Ventana puts a spectrum of options at her disposal. Alice could do nothing and assume that Bob and Carl will act responsibly. She could scan VMs for the insecure binary and email their owners (she can even check up on them later). She could patch the insecure binaries herself. Finally, she has many options for denying access to copies of the insecure binary: use a server file ACL to deny reading or executing it, use a Ventana version ACL to prevent reading it even as the older version of a file, use a branch ACL to deny any access to the branch that contains it (perhaps appropriate for long-unused branches), and so on. Alice can take these steps for any file stored in Ventana, whether contained in a VM that is powered on or off or suspended, or even if it is not in any VM or view at all.

Third, once the immediate problem is solved, Alice can work to prevent its future recurrence. She can configure a malware scanner to examine each new version of a file added to Ventana as to whether it is the vulnerable program and, if so, alert Alice or its owner (or take some other action). Thus, Alice has reasonable assurance that if this particular problem recurs, it can be quickly detected and fixed.

5.5.2 Benefits for Widgamatic

We now consider how Alice, Bob, Carl, and everyone else at Widgamatic benefit from using Ventana instead of virtual disks. We use virtual disks as our main basis of comparison because Ventana's advantages over conventional distributed file systems are more straightforward: they are the versioning, isolation, and encapsulation features that we intentionally added to it and have already described in detail.

Central Storage

It's easy for Bob or Carl to create virtual machines. When virtual disks are used, it's also easy for Bob or Carl to copy them to a physical machine or a removable medium, then lose or forget about the machine or the medium. If the virtual machine is rediscovered later, it may be missing fixes for important security problems that have arisen in the meantime.

Ventana's central storage makes it more difficult to lose or entirely forget about VMs, heading off the problem before it occurs. Other dedicated VM storage systems also yield this benefit [87, 74].

Looking Inside Storage

Alice's administration tasks can benefit from "looking inside" storage. Consider backup. Bob and Carl want the ability to restore old versions of files, but Alice can easily back up virtual disks only as a collection of disk blocks. Disk blocks are opaque, making it hard for Bob or Carl even to determine which version of a virtual disk contains the file to restore. Doing partial backups of virtual disks, e.g. to exclude blocks from deleted temporary files or paging files, is also difficult.

File-based backup, partial backup, and related features can be implemented for virtual disks, but only by mounting the virtual disk or writing code to do the equivalent. In any case, software must have an intimate knowledge of file system structures and must be maintained as those structures change among operating systems and over time. Mounting an untrusted disk can itself be a security hole [88].

On the other hand, Ventana's organization into files and directories gives it a higher level of structure that makes it easy to look inside a Ventana file system. Thus, file-based backup and restore requires no special effort in Ventana. (Of course, in Ventana it is natural to use versioning to access file "backups" and ensure access by backing up Ventana servers' storage.)

Sharing

Sharing is an important feature of storage systems. Bob and Carl might wish to collaborate on a project, or Carl might ask Alice to install some software in his VM for him. Virtual disks make sharing difficult. Consider how Alice could access Carl's files if they were stored on a virtual disk. If Carl's VM were powered on or suspended, modifying his file system would risk the guest OS's integrity, because the interaction with the guest's data and metadata caches would be unpredictable. Even reading Carl's file system would be unreliable while it was changing, e.g. consider the race condition if a block from a deleted directory was reused to store an ordinary file block.

On the other hand, Ventana gives Alice full read and write access to virtual machines, even those that are online or suspended. Alice can examine or modify Carl's files, whether the VM or VMs that use them are running, suspended, or powered off, and Bob and Carl can work together on their project without introducing any special new risks.

Security

If Widgamatic's VMs were stored in virtual disks, Alice would have a hard time scanning them for malware. She could request that users run a malware scanner inside each of their VMs, but it would be difficult for her to enforce this rule or ensure that the scanner was kept up-to-date. Even if Bob and Carl carefully followed her instructions, VMs powered

on after being off for a long time would be susceptible to vulnerabilities discovered in the meantime until they were updated.

Ventana allows Alice to deploy a scanner that can examine each new version of a file in selected branches, or in all branches. Conversely, when new vulnerabilities are found, it can scan old versions of files as well as current versions (as time is available). If malware is detected in Bob's branch, the scanner could alert Bob (or Alice), delete the file, change the file's permission, or remove the virus from the file. (Even in a private branch, files may be externally modified, although it takes longer for changes to propagate in each direction.)

Ventana provides another important benefit for scanners: the scanner operates in a protection domain separate from any guest operating system. When virtual disks store VMs, scanners normally run as part of the guest operating system because, as we've seen, even read-only access to active virtual disks has pitfalls. But this allows a "root kit" to subvert the guest operating system and the malware scanner in a single step. If Alice runs her scanner in a different VM, it must be compromised separately. Alice could even configure the scanner to run in non-persistent mode, so rebooting it would temporarily relieve any compromise, although of course not the underlying vulnerability.

A host-based intrusion detection system could use a "lie detector" test that compares the file system seen by programs running inside the VM against the file system in Ventana to detect root kits, as in LiveWire [89].

Access to Multiple Versions

Suppose Bob wants to look at the history of a document he's been working on for some time. He wants to retrieve and compare all its earlier versions. One option for Bob is to read the old versions directly from older versions of the virtual disk, but this requires accurate interpretation of the file system, which is difficult to maintain over time. A more likely alternative for Bob is to resume or power on each older version of the VM, then use the guest OS to copy the file in that old VM somewhere convenient. Unfortunately, this can take a lot of time, especially if the VM has to boot, and every older version takes extra effort.

With Ventana, Bob can attach all the older versions of his branch directly to his view. After that, the different versions can be accessed with normal file commands: `diff` to

view differences between versions, `grep` to search the history, and so on. Bob can also recover older versions simply by copying them into the his working branch.

Chapter 6

Evaluation

The previous chapters described the design of our extreme paravirtualization and virtualization aware file system prototypes. This chapter evaluates these prototypes for performance, code size, and portability. We evaluate the network and file system paravirtualization prototypes separately from the Ventana virtualization aware file system prototype.

6.1 Extreme Paravirtualization

This section compares PON and POFS micro- and macro-benchmarks against conventional paravirtualization approaches. We show that the performance penalty is minimal (under 4%) in most cases, and even demonstrate speed-ups in important special cases. We show how the multi-core CPU architecture contributes to this performance. We also report on the size of the code to implement PON and POFS.

We used Xen, a free software hypervisor, to obtain the numbers reported in this section. Xen provides the inter-VM communication primitives needed by PON and POFS "out of the box": static shared memory and page sharing via Xen *grant tables*, and remote interrupts via Xen *event channels*. However, Xen is designed to provide these communication primitives between application VMs (called *DomU*s) and a privileged VM (called *Dom0*), not between different DomU VMs, so we implemented a new "virtual shared memory" Xen device that provides a rendezvous point for VMs to connect with each other.

Our test machine was configured with 8 GB RAM. It ran Xen "unstable" in 32-bit PAE

mode. Both privileged (Dom0) and user (DomU) VMs used Debian GNU/Linux "unstable" as operating system, using Linux kernel 2.6.18, which was the latest kernel with an official Xen patch set at time of testing. Each user VM was configured with 768 MB RAM, one 4 GB pre-allocated virtual disk, and one virtual CPU. We also supplied 1 GB swap to each VM, although our tests did not exercise it.

Our VMs used the `ext3` file system on physical and virtual disks. Reported timings are minimum observed values across multiple runs.

6.1.1 Processor Configuration

Our test machine was configured with a pair of dual-core 2.33 GHz Intel Xeon 5140 processors. This reflects the start of a trend for commodity servers, and to a lesser extent commodity desktop machines, toward an increasing number of processor cores. Quad-core processors are available today to give a 4-CPU server a total of 16 cores. In coming years the number of cores will continue to increase: Intel, for example, has predicted the availability of 80-core CPUs by 2011, according to CNET [90].

There are as yet few operating systems and applications that makes good use of these CPU resources. Our results show one way to take advantage of multi-core CPUs to reduce performance overheads traditionally seen for communication on a single-processor machine. We believe that the ability to quickly schedule a decomposed piece of an operating system will mean that the overheads of communication will be manageable, particularly in comparison to the benefits.

Supporting this claim, the measurements reported in the following sections, which were taken with each virtual CPU assigned dedicated use of a physical processor core, reflect considerable improvement over single-processor results. This is most noticeable in the networking tests, in which the timings increase by as much as 100% when only a single core is used.

This argument in favor of dedicating a core to a service such as a networking stack or a file system is reasonable when cores are not already put to good use, but it is less attractive when they are already in high demand. The use of virtual machines for server consolidation is an example of the latter, because an additional core can always be used to

run an additional virtual machine.

For two reasons, however, we believe that this need not be a barrier to deployment of extreme paravirtualization. First, extreme paravirtualization lends itself to going a step beyond server consolidation, to service consolidation, because one device module on a core can serve a large number of VM clients. Thus, the demand for cores is not doubled or tripled but merely increased slightly. Furthermore, dedicating a core to a device may significantly reduce CPU activity in other VMs, e.g. a dedicated network device module handles all TCP timers on behalf of its VM clients, reducing the interrupt load on those clients. In turn, this allow more VMs to run, and the result may be neutral or even a net gain in terms of processor efficiency.

Second, in situations where large number of clients should not share a single device module VM due to e.g. security or reliability concerns, some of the benefits of dedicated cores may be obtainable through coscheduling, that is, scheduling a service VM and its client VMs at the same time on separate cores [91]. The unit of scheduling becomes a collection of VMs, instead of a single VM.

We compared our prototype's performance against that of Xen's paravirtual Ethernet and virtual disk implementations. Xen, like Microsoft's Viridian hypervisor, runs device drivers in a separate VM that must be promptly scheduled to get high I/O performance. This architecture is designed to take advantage of the trend to multicore CPUs, so to ensure highest performance we dedicated a physical processor core for the virtual CPU of Dom0 that contains the device drivers.

Our approach would also benefit from a similar optimization so that the service VM with the paravirtualized device and device driver would be promptly scheduled. Unfortunately our Xen prototype service VM was not able to talk directly to the device and required going through the driver Dom0. To avoid the introduction of context switches where the real implementation would have none, we included an additional core for running the paravirtualized device's VM. Although we are using a total of three cores we do not believe we get any additional benefit over what the architecture would show with only two cores. In fact, having to communicate with Dom0 to access the I/O device likely worsened our performance compared to direct device access.

6.1.2 Network Paravirtualization

This section compares our network extreme paravirtualization prototype against conventional paravirtualization. First, we show that PON's performance penalty for communicating over a TCP/IP network is at most 3% in bandwidth and latency. We also show that, for the special case of communicating with a VM on the host, PON actually improves bandwidth by up to 294% and reduced latency by up to 207%. Second, we show that this is achieved with code much smaller than typical TCP/IP stacks.

TCP/IP Network Performance

We ran bandwidth and latency micro-benchmarks to compare PON and TCP/IP performance. We used a VM with a conventionally embedded TCP/IP stack as our basis for comparison. Our PON prototype was set up as shown in Figure 4.2 on page 28, connecting an application VM to a PON paravirtualized network gateway VM. This allows the application VM, which does not need to contain a TCP/IP implementation, to interact with a TCP/IP network.

We implemented the gateway as a Linux kernel module that connects PON to the Linux TCP/IP stack. To reduce the trusted code base to the minimum, production versions of a gateway VM might be based on a stripped-down operating system. We configured networking identically in the gateway VM and the VM with conventional networking, except for the presence of the gateway module.

The remote network target we used for these tests had hardware identical to our primary test machine and ran Linux on bare hardware. It was directly connected to our primary test machine over 1-Gb Ethernet, without an intervening hub or switch.

With each setup, we measured the time to transfer 100 MB of data to our second physical machine, using a pair of simple programs that call `poll` to wait for data to arrive or for buffer space to become available and then `read` or `write` to send or receive data. We varied both the number of streams among which data was divided and the size of the buffer passed to `read` or `write` in a single system call, which we call the *chunk size*. We ran the test for each combination of 1, 4, and 16 streams with 16-byte, 64-byte, 4-kB, and 64-kB chunk sizes.

Streams	Protocol	Read and Write Chunk Size			
		16 byte	64 byte	4 kB	64 kB
1	PON	3.782 s	1.114 s	.120 s	.115 s
	TCP	4.580 s	1.453 s	.134 s	.133 s
4	PON	3.757 s	1.113 s	.127 s	.116 s
	TCP	4.401 s	1.453 s	.137 s	.131 s
16	PON	3.783 s	1.118 s	.132 s	.125 s
	TCP	4.281 s	1.415 s	.521 s	.299 s

Table 6.1: Time to transfer 100 MB of data with PON and Linux TCP, for varying numbers of parallel streams and read/write chunk sizes.

We also ran a second simple test to measure the latency of interactive protocols on each setup. This test passes a 4-byte data segment back and forth across a connection as fast as possible, repeating 100,000 times. A segment is transmitted as soon as the previous one is received. Each segment simply contains an integer that is incremented on each succeeding transmission to verify integrity. To ensure minimum latency, we disabled the Nagle algorithm and TCP delayed acknowledgements.

In every case, our results show that PON has minimal impact. In both bandwidth and latency, and in the bandwidth tests, for every tested combination of chunk size and number of streams, PON had a 3% or smaller performance penalty, compared to the conventional TCP/IP setup. We attribute this to reduced overhead in the PON gateway VM versus the conventional setup: the gateway runs entirely within the kernel, without the overhead due to system calls incurred by the conventional VM network stack.

To ensure that improvements in bandwidth and latency do not come at the expense of CPU time, we also measured the CPU usage of PON versus Linux TCP. CPU usage of PON was generally similar to, or slightly lower than, that of Linux TCP, except that it was spread across two CPU cores instead of one.

Inter-VM Performance

In the special case where a pair of virtual machines on the same host wish to communicate with one another, a PON link can connect them directly, with no need for a conventional

network stack at all. We measured PON's performance against the Linux TCP/IP implementation for this special case of data transfer.

Table 6.1 shows the time, in seconds, for each tested combination of number of streams and chunk size, for PON and TCP/IP bulk transfers. In each case, PON achieves between 11% and 294% faster bulk transfers than TCP, despite PON's simplicity. This is primarily due to performance tuning specific to the PON environment (described in detail below). Another factor is how PON shares memory directly between VMs, whereas TCP/IP uses the Xen privileged VM (Dom0) as a trusted third party to pass around the pages containing Ethernet frames.

As for latency, we measured PON to run our latency test in .918 s and Linux TCP in 2.814 s. The TCP results seem to be primarily due to the extra level of indirection in routing Ethernet frames through Xen's Dom0: when we perform the test on Linux TCP against localhost within a single VM, the Linux TCP time drops to .994 s.

As a fraction of their runtime, PON and TCP use about the same amount of CPU time. Because PON runs faster than TCP, the total amount of CPU time needed to transfer a given amount of data is much less with PON.

PON streams also save memory compared to TCP implementations, because data sent over a PON stream does not need to be buffered by the sender to guard against data loss. The PON prototype uses a fixed 64-kB buffer for each connection, because we found that further increasing the buffer size caused little or no increase in bandwidth, although although dynamically sized buffers could be implemented. In contrast, the Linux TCP implementation in its default configuration scales up the size of the buffers in some of the tests to multiple megabytes, although limiting its buffers to 64 kB has little impact on its performance.

Tuning In implementing and tuning our PON implementation, we discovered a number of keys to performance. By far the most important of these is limiting the number of remote interrupts exchanged between the communicating VMs. Remote interrupts are implemented at the hardware level as inter-processor interrupts (IPIs). For small chunk sizes, sending an IPI per chunk sent or received makes data transfer far too expensive, e.g. the

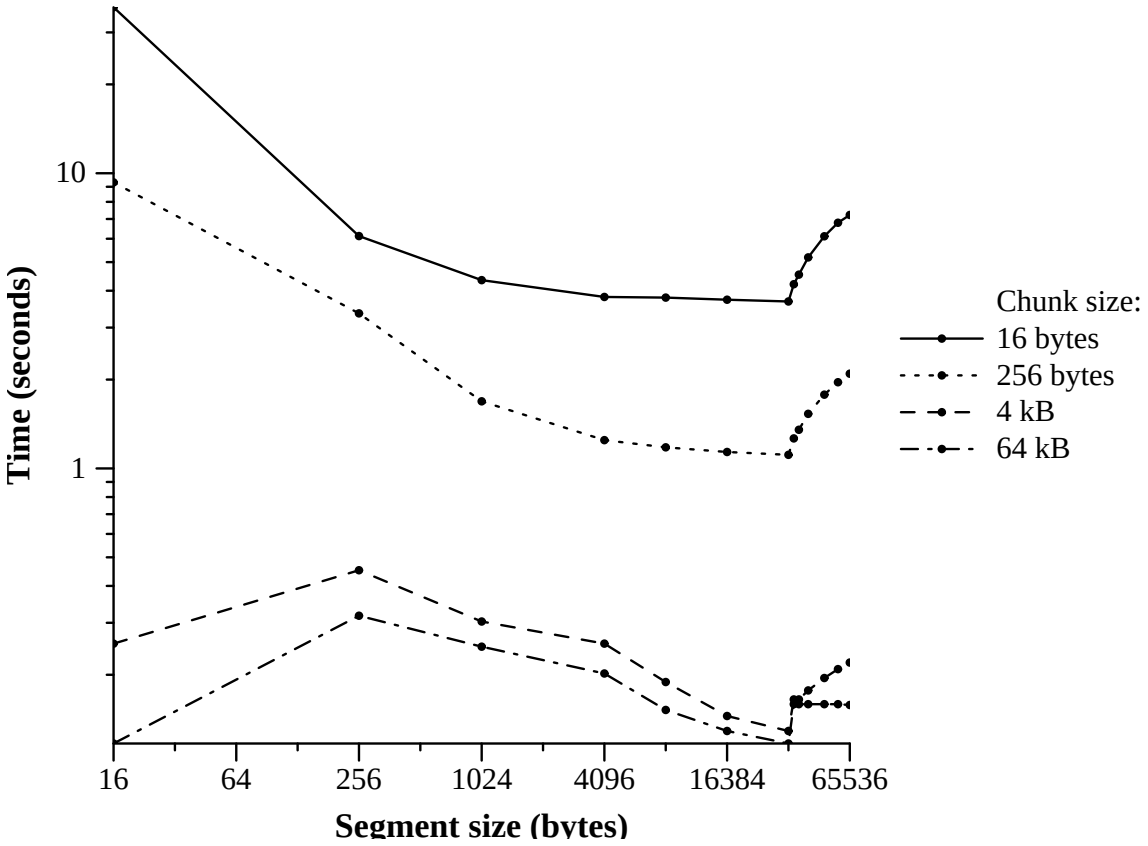

Figure 6.1: Time to transfer 100 MB using a single PON stream, versus PON segment size, for varying chunk sizes.

time to transfer 100 MB in 16-byte chunks increases 864% with a single stream, and similarly with more streams. On the other hand, sending an IPI only when the 64-kB buffer is filled also increases transfer time (by about 90%), because it prevents any overlap in processing: the receiver is only able to start reading data when the sender has completely filled the buffer, and vice versa.

Thus, we adopted the idea of a virtual "segment size," a count of bytes sent or received before sending a remote interrupt. A segment size equal to the chunk size is equivalent to sending one IPI per chunk, a 64-kB segment size is equivalent to sending an IPI only after filling the buffer, and sizes in between represent a spectrum of intermediate policies.

Figure 6.1 plots segment size versus transfer time for each chunk size in Table 6.1. The graph shows that a 32 kB segment size yields the minimum transfer time for all tested

chunk sizes. Thus PON uses a 32 kB segment size by default, and all other measurements in this section were taken at this segment size.

An important secondary factor in performance is minimizing cache line bouncing between processor cores. Updating the send and receive buffer pointers in shared memory once per chunk, for example, instead of once per segment, increases total transfer time at small chunk sizes by up to 34%.

PON does not achieve its performance by reducing the number of times data is copied. In fact, it copies data the same number of times as the Linux TCP implementation. PON does two copies: the sender copies data into a shared segment and the receiver copies data out of it of it. Linux TCP implementation does the same with Ethernet frames, which are passed from VM to VM using Xen page transfer primitives. A number of ways to avoid copying entirely for network I/O have been proposed [92], but we did not implement them in our prototype; the best techniques require changes to applications, which our present work avoids.

Furthermore, data copying is less costly than one might expect. Presumably, the linear memory access pattern in the shared buffers is easily predicted by the CPU cores; we did not use explicit prefetching to improve performance. With a 16-byte chunk size, profiling (using Xenoprof [93]) shows that most of the cost of data transfer is system call and synchronization overhead, and omitting data copying entirely (by commenting out the code to copy data to and from the shared buffer) improves performance by 10% or less. With 64-kB chunks, the same experiment improves performance by about 85%, but this is only about 53 ms faster for the 100 MB transfer.

Code Size

Our code for PON consists of a 2,000-line library that implements the PON protocol, plus a 1,300-line set of wrappers that implement a Linux protocol family to make the PON protocol available to user applications through the familiar socket interface. Thus, PON does not add greatly to the size of the operating system. We believe that PON's simplicity and small size reflect positively on its potential for security.

PON adds much less code to an operating system than would a network stack. For example, the Linux 2.6.18 implementation of IPv4, TCP, and UDP, not including header

files, optional features, or core socket code, consists of more than 27,000 lines. Even lwIP, a simple TCP/IP stack for embedded applications [94], contains over twice as many lines of code as PON.

6.1.3 File System Paravirtualization

As for the paravirtual network implementation, we evaluate our paravirtual file system on the basis of performance and code size.

Performance

We compared the performance of our file system implementation against virtual disk-based file systems and network file systems. To compare against a virtual disk-based file system, we used a file system on a Xen virtual block device. To compare against a network file system, we used the Linux in-kernel NFS server and client running in a virtual machine on the same host. Many Linux NFS parameters can be tweaked to improve performance (UDP vs. TCP, read and write buffer size, asynchronous vs. synchronous writes, etc.) so we tried a variety of configurations and reported, for each test, the best performance among the set of configurations.

We configured POFS and the NFS server to serve a single, initially empty directory to the client from a server VM virtual disk. For the virtual disk tests we used the same virtual disk that stored other client VM data. All file systems were formatted as `ext3`.

Linux NFS is our primary performance target. POFS should perform better than NFS, because NFS is designed to cope with lossy networks by buffering a copy of written data in the client until it reaches disk, and with unreliable server software and hardware by synchronously writing data to disk. It also requires more data copying than either virtual disks or POFS.

The speed of operations in the virtual disk case represents an upper bound for POFS performance, as POFS performs the same operations but with the cost of inter-VM communication added. Because each file system operation must wait for a response from the server before the client's system call can return a success or failure code, POFS performance is not significantly aided by parallelism between processors in the way that PON

Storage System	Build	Create	Extract
Virtual Disk	28.6 s	.14 s	.15 s
POFS	29.7 s	.47 s	.26 s
Linux NFS	31.4 s	1.67 s	.34 s

Table 6.2: File system performance testing results.

performance was aided. (`read` and `write` system calls where the data is already mapped and the file size does not change are exceptions.)

We ran the following tests:

Build Build GNU `tar` within the tested storage system. This is a macro-benchmark that tests a mix of file system operations.

Create Copy the `/dev` hierarchy from a virtual disk to the tested storage system. This hierarchy contains 5,567 files spread among only 13 directories, and very little data. Thus, this is a micro-benchmark of file creation performance.

Extract Extract the GNU `tar` program source code from a 2.6 MB compressed `tar` archive on a virtual disk into the tested storage system. This is a micro-benchmark of file write performance.

Table 6.2 reports our performance results. We measured times in the guest OS using the `bash` shell's `time` command.

The results show that POFS consistently performs much better than NFS, especially for the file creation micro-benchmark. Furthermore, in the macro-benchmark Build test, POFS lagged behind a virtual disk based file system by less than 4%. For the Create and Extract micro-benchmarks, POFS did lag significantly behind virtual disks, although it was still much faster than Linux NFS in its fastest configuration. We expect that we could improve micro-benchmark performance significantly with some optimization effort.

POFS used less CPU time than NFS. This was most pronounced in the Create benchmark, in which POFS used an average of 8.6% of the client CPU and 10.4% of the server CPU, whereas NFS used 27.4% and 28.7%, respectively.

Tuning Our measurements reflect one significant result of performance tuning, in which the server CPU core runs in a loop that continually polls its clients' request rings, instead of using interrupts to wake up the server. Without this optimization, the Create test takes .23 s longer, the Extract test .11 s longer, and the Build test .20 s longer. Because the file server VM is used only for file service, and because its virtual CPU has a dedicated physical processor core, polling does not degrade other VMs' performance.

Using polling also in the client VM, to wait for responses to server requests, yields a significant further increase in performance, but only at the expense of other processes running in the client VM, so our measurements do not reflect client-side polling.

Code Size

Our implementation of the POFS client is a single C file under 1,000 line long. Thus, POFS adds a minimal amount of code to the operating system. We believe that POFS's simplicity and small size reflect positively on its potential for security.

PON adds much less code to an operating system than would an ordinary local or network file system. The Linux `ext2` local file system, for example, is over 5,000 lines of code, even without support for special features such as ACLs and extended attributes. The `9p` file system, the smallest Linux network file system, is over 6,000 lines of code.

6.1.4 Portability

PON and POFS depend only on a few low-level communication primitives, which should be implementable in a wide variety of VMMs. We used Xen as our environment for performance testing, but we also created two other implementations of our shared memory library for use in different environments, which helps to demonstrate the potential for portability of extreme paravirtualization design among VMMs. All three implementations share the same function interface, so that code that uses it for the most part need not be aware of which version it is using. This variety of implementations helped to assure us that our primitives are widely supportable.

Our second implementation builds the shared memory primitives out of inter-process communication system calls: `mmap`, `pipe`, etc. Thus, it offers these primitives to ordinary

user processes running in Linux, with no need for a VMM environment. This implementation provided a convenient, friendly platform for initial debugging and profiling.

The third implementation runs under QEMU [95], a virtual machine monitor that runs as an ordinary user program on top of a host operating system. This version presents shared memory to the guest OS as virtual PCI cards whose RAM can be mapped into the virtual machine's "physical" RAM. Remote interrupts correspond to virtual PCI interrupts. QEMU, in turn, uses the POSIX version of the library to obtain the shared memory provided to the guest OS. QEMU slows code execution by a variable factor between 4 and 20, making this version's performance particularly difficult to evaluate.

6.2 Ventana

To demonstrate that the approach taken in Ventana, our virtualization-aware file system, can yield reasonable performance, we ran tests comparing Ventana's performance against virtual disks and two conventional NFSv3 servers. Before we began testing, we considered what we expected to see for relative performance of the four storage choices:

- Virtual disks should yield the highest absolute performance. Disks are not shared media, so their contents can be cached indefinitely by the guest OS. Operating systems are carefully designed to minimize the impact of disk latency.

- The Linux 2.6.12 kernel-mode server, one of the conventional NFSv3 servers we tested, should yield the next highest performance. The kernel server has the inherent advantages over the other two NFSv3 servers that it can bypass system call and context switching overhead. According to copyright notices, its development began in 1996, making it old enough to have been optimized and tuned. The kernel server also uses hand-coded XDR handlers that allow it to encode and decode NFSv3 messages faster than the other two servers.

- `unfs3` 0.9.13 [96], a user-mode NFSv3 implementation, should yield the next highest performance. Like Ventana, it is a user program, so it is also subject to system call overheads, etc. This server had been in development for about two years when we

Storage	Boot	Create	Extract	Build
Virtual Disk	64	2	2	172
Linux NFS	78	59	12	229
`unfs3`	130	*	10	239
Ventana	131	45	11	239

Table 6.3: Ventana performance testing results. Entries are times in seconds. *Could not complete test.

ran our tests and appeared to be roughly a "beta" release. It has been in development much longer than Ventana, so it should be better tuned and therefore faster.

- Ventana is likely to be the slowest NFSv3 server for the reasons above. Some basic optimizations have been applied to Ventana, but it has not been extensively optimized or tuned.

We used a single machine for testing, an IBM ThinkPad T30 with a 1.8 GHz Pentium 4 processor and 1 GB RAM. The NFSv3 servers ran on the host. The client ran inside a virtual machine, using VMware Workstation 5.0 as the virtual machine monitor [97]. The client's VM was configured with 64 MB RAM, one virtual Ethernet card, and one 4 GB sparsely allocated virtual disk. Both guest and host ran Debian GNU/Linux "unstable" with a Linux 2.6.12 kernel. The guests and hosts used the ext3 file system on their disks. The guest's disk was configured with an IDE interface. We disabled non-essential services on the guest and host to minimize variation between runs.

We configured the conventional NFS servers to serve a single, initially empty, directory to the client. Similarly, we configured Ventana with a view of an initially empty private branch. In all cases, communication between the client and server was over "localhost," eliminating the delays of real networks as a variable.

We ran the file system tests already described in Section 6.1.3, plus the following test:

Boot Time to boot using the tested storage system, from the boot loader prompt to the Linux login prompt. For the NFS servers, this means configuring Linux to boot from an "NFS root" file system. This tests read performance.

6.2.1 Results

Table 6.3 reports our performance results, obtained by running each test multiple times and reporting the mean. The Boot test was timed manually by watching a clock. Times for other tests were measured in the guest OS using the `bash` shell's `time` command.

The results are largely as expected: in general, virtual disks are fastest, followed by Linux NFS, `unfs3`, and Ventana, in that order. However, in the Create test, Ventana is significantly faster than the Linux NFS server. We discovered that this is because Linux synchronously creates files, which is not explicitly required by NFSv3. In the Extract test, `unfs3` and Ventana are slightly faster than Linux NFS for the same reason.

The `unfs3` server was unable to complete the Create test. In multiple attempts, it reported many "stale file handle" errors and eventually hung.

6.2.2 Analysis

As we tuned the NFS servers during testing, we found that two factors dominate NFSv3 performance: primarily, the speed of COMMIT, and secondarily, read and write block size.

COMMIT is a NFSv3 RPC that synchronously forces a file's inode and contents to disk. An NFSv3 server must not reply to a COMMIT call until the file's inode and contents are on disk. Thus, COMMIT is analogous to `fsync` in a POSIX system. A typical NFS client, such as the Linux 2.6.12 kernel client, writes files with a series of asynchronous WRITE requests, followed by a single COMMIT request when the process that wrote the file closes it. The COMMIT request sent on close can easily cause NFSv3 performance to bottleneck on COMMIT performance.

We observed this in practice. An early version of Ventana disregarded COMMIT requests, replying immediately. This version completed the Build test in 216 seconds (about 10% faster), making it the fastest of the NFS servers. Our next attempt checkpointed all modified files and the version log to disk on a COMMIT call. This version took over 360 seconds to complete the Build test. Finally, we implemented a selective COMMIT that only forces the specified file to disk, which completed the Build test in 239 seconds, the figure reported in Table 6.3.

We found block size, that is, the maximum number of bytes that may be read or written

in a single READ or WRITE call, to be a secondary factor in NFS performance. The times reported in Table 6.3 were all produced using a 32 kB block size. (By default, `unfs3` limits block size to 8 kB, but we overrode this value by modifying its source code.) Reducing block size to 8 kB increased the time for the Build test by about 6% for Ventana, but 1% or less for the other NFSv3 servers. The current implementation of Ventana has high per-call overheads because it does not cache file descriptors from call to call, so we speculate that this is the reason for the penalty.

The penalty for smaller block sizes is not due to synchronous writes. NFSv3 clients do not typically request synchronous writes. Instead, they make asynchronous WRITE calls followed by a synchronous COMMIT, as already described.

6.2.3 Potential for Optimization

Two kinds of optimization are possible. The first is speeding up Ventana's NFSv3 server. Currently Ventana stores each file version as two files in the host file system: one for the inode, one for contents. A COMMIT call must `fsync` both. Using a log to store file and inode changes could reduce the `fsync` calls to just one. A log file would likely be stored contiguously on disk, unlike the many files that require `fsync` in the current approach. These factors together would likely speed up COMMIT significantly. Ventana could also cache file descriptors from one call to the next.

But even an excellent implementation of an NFSv3 server is unlikely to exceed the speed of the Linux in-kernel server, given existing NFSv3 clients. For Ventana performance to approach the speed of a virtual disk, we believe that protocol changes, and thus modification of clients, will be necessary. Support for NFSv3 could be retained for compatibility with guests that don't have special support for Ventana.

Thus, a more advanced implementation would, for performance, want to implement a custom protocol, such as POFS, or at least an extended version of NFS, for communication between the guest OS and Ventana's host manager. This is our second kind of optimization. Because both the guest and the host manager run on the same physical machine, they can make assumptions that conventional network file system clients and server cannot. For example, the client and server both have the same endianness, so there is no need

for them to convert to and from network byte order. Perhaps most importantly, the guest OS and the host manager can use shared memory to transmit file blocks, instead of using network protocols that require copying. This approach could yield performance approaching that possible with virtual disks, as shown by POFS and other inter-virtual machine file systems [71].

Chapter 7

Related and Future Work

This chapter discusses related and future work, organized by category. Work related to extreme paravirtualization is broken into sections that cover designs used for modular operating systems, virtual machine monitors, network stacks, and distributed file systems. Virtualization aware file systems are treated separately. A final section covers ideas for future work.

7.1 Modular Operating Systems

The result of applying the extreme paravirtualization approach to both the network and block storage of an modern operating system is shown in Figure 7.1. This structure closely resembles that of the research microkernel operating systems of the 1980s. In fact, many of the benefits we claim in this thesis, such as improved OS modularity and the ability to easily substitute alternative implementation for kernel components, were among the selling points for microkernels. We claim that the extreme paravirtualization approach has the ability to achieve these benefits in an evolutionary way that leverages existing industry trends and bypasses the problems that impeded widespread adoption of microkernels.

Dominant modern operating system environments such as Microsoft Windows, Linux, and Mac OS X resemble the monolithic kernel structure more than they do microkernels, so it is clear that in spite of its benefits, the microkernel approach suffered from some problem that caused it to fail to influence the industry away from the monolithic kernel

approach. Performance concerns are commonly mentioned as a cause of this failure, but it fundamentally boiled down to the inability to convince OS kernel designers and maintainers to adopt the approach.

Unlike 1980s microkernels, where claims of a better OS structure competed against performance problems and inertia against change, the virtualization layer is already being deployed by the industry for its reduction in management cost and better resource utilization. Even monolithic OS vendors have embraced virtualization, with Microsoft touting its Viridian hypervisor [62] and the multiple virtualization approaches being promoted in the Linux community [48, 98]. Our approach can assume the existence of the hypervisor as a starting point and evolve the decomposed architecture from there.

Rather than having to convince OS kernel designers to adopt our approach, paravirtualization at existing OS interfaces allows our approach to proceed without help from the kernel designers. Our experience with the Linux kernel shows how our approach can be done with no changes to the existing operating system kernel and using inter-VM services common to modern hypervisors.

We believe that the ability to form this decomposition in a piecemeal way will mitigate any performance concerns. Furthermore, workloads in virtual machines that suffer from the overheads can always choose to not use the extreme paravirtualized interfaces, leaving only the applications that find enough benefit from them to use it. This improves on the all-or-nothing approach offered by microkernels.

7.2 Virtual Machine Environments

Parallels Desktop 3.0 and VMware Workstation 5.0 virtualize 3-D graphics in an extreme paravirtualization-like manner [99, 100]. Instead of providing an interface that resembles hardware 3-D interfaces, these products provide an implementation of common 3-D APIs (DirectX and OpenGL) that is then passed through to the host's implementation of these APIs. This approach is likely motivated by the undocumented, rapidly changing, and difficult-to-translate nature of real 3-D hardware interface, which differs considerably from our motivations.

LeVasseur et al. [101] showed how drivers for physical devices can be moved into

separate VMs. It provides a way to improve the reliability of a VMM by moving its physical device drivers into separate modules, whereas our work aims to improve the modularity (and hence reliability) of the operating systems inside VMs.

Several VMMs have implemented inter-VM memory sharing. IBM's z/VM has Discontiguous Shared Segment (DCSS) support, a feature dating back at least as far as CP-67, which allows a read-only, shared segment of storage to be loaded at the same address in multiple VMs, to reduce host memory requirements when multiple VMs have common code or data [102]. VMware ESX Server dynamically detects identical memory pages across VMs and merges their contents, with copy-on-write if one copy is changed [103]. The Xen VMM also has inter-VM communication primitives, as discussed in Section 6, that have been used to implement fast inter-VM RPC [104, 71]. Related mechanisms were supported in CP-67 since at least 1979 [65].

Adding secure message passing and memory sharing to VMMs has been proposed as a way of reducing the total size of the trusted computing base of a system, by making it possible to use untrusted components as part of a trusted system [105, 106]. As a simple, fast, secure protocol, PON may be an appropriate basis for such a communication system.

Warfield's Ph.D. thesis [107] also addresses the implementation of virtual devices in a virtual machine environment. Warfield's approach does not raise the virtual interface's level of abstraction.

Linux-VServer [108] uses an OS-level virtualization approach to allow one Linux kernel to act as multiple virtual servers. It is superficially similar to our prototypes, which also divide a system into multiple Linux kernels. However, Linux-VServer does not pull any functionality out of the main kernel into virtual servers. Also, OS-level virtualization provides fundamentally less isolation between VMs than does a traditional VMM. Finally, our prototype is only Linux-based for convenience; a production version of PON, for example, would likely be based on a simpler system, for security.

The Proper system designed for PlanetLab [109] provides a means for VMs in a system virtualized at the system call level under Linux-VServer to request privileged operations from another VM. It is a much higher-level system than the one we propose here, because it can take advantage of features of the OS shared among VMs; e.g. Proper uses file descriptor passing among processes as an important building block.

Plan 9 [110] is based on a protocol, called `9p`, that can be used for general-purpose access to and manipulation of local and remote resources [111]. `9p` could be used as the basis of PON, POFS, and other kinds of extreme paravirtualization. However, as a network protocol, `9p` cannot take advantage of shared memory as can protocols designed for it, such as PON and POFS.

μDenali [112], by Whitaker et al., is a paravirtualizing VMM built on the concept of allowing a parent VM to provide services to child VMs by interposing on the child VM's virtual device interactions. It is not clear whether this interposition mechanism generalizes to higher-level interfaces; the services described in the μDenali paper operate at the same level of abstraction as ordinary hardware.

Self-virtualizing devices, that is, devices with multiple hardware contexts that can be assigned to and used directly by VMs, have been proposed to increase network scalability [113], an idea closely related to user-accessible network interfaces [61, 114, 115]. An extreme paravirtualization device module could use the increased performance of self-virtualizing devices or, possibly, user-accessible network devices, in the same way as a conventional VM device architecture.

The Collective manages VMs whose data is cached on portable storage devices carried by users [116]. Data for VMs in the Collective is maintained in the form of virtual disks. Use of a VAFS could allow the Collective to more effectively choose data for caching and backup, because of the additional structure imposed by a hierarchical file system compared to a virtual disk. The GVFS and Internet Suspend/Resume projects that migrate virtual machines across a network could similarly benefit [117, 118].

7.3 Network Paravirtualization

Much research has explored the different ways that a network stack can be divided among hardware and software components. Multiple researchers have explored dedicating one or more processors, processor cores, hardware threads, or cluster nodes to network processing [119, 120, 121, 122]. Like PON, many of these systems use a shared memory interface to the network stack. These systems focus on performance, however, and their network stacks are not isolated from the rest of the operating system.

User-level networking moves most network processing from the kernel into user applications. This can speed up networking by reducing context switches [123, 69, 114] or by allowing network implementations to be specialized to application needs [92, 124, 125]. Each PON gateway VM is essentially a user-level network stack and could take advantage of some of the same hardware features. User processes in VMs could also interface to a PON link directly, bypassing the guest kernel. This would require modifying user applications, which our present work avoids.

The Mach microkernel demonstrates another way to move network processing to user space, by moving an entire monolithic Unix kernel into user-space [51]. Mach also showed that performance could be comparable to an in-kernel implementation [126].

In hardware, TCP offload engines (TOEs) seek to accelerate networking by implementing a network stack in hardware [127, 128, 129, 130], analogous to how PON implements a network stack in virtual hardware. However, because they are implemented as specialized hardware, they cannot offer the same flexibility as PON. They often suffer from low resource limits, e.g. Adaptec TOEs are limited to 1,024 simultaneous TCP connections [127]. Unlike PON, TOEs can be difficult to upgrade or not upgradable, and difficult to adapt to new protocols. Furthermore, the PON protocol is simple, but a TOE often has a complex interface, e.g. Microsoft TCP Chimney [131], with associated overhead that can be high enough to negate the TOE's advantages [132, 133].

The PON approach of using a socket-level proxy (the PON gateway) to access an external network has been applied to wireless links of varying speed. In a wireless environment, the lossy quality of the link adversely affects performance, because TCP interprets loss as congestion. Using a wired proxy avoids the issue [134, 135].

Fast Sockets [136], like PON, transparently replaces TCP/IP with another protocol within a given domain, in its case among hosts on a Myrinet network. Unlike PON, the primary goal of Fast Sockets is to reduce latency and increase bandwidth. Fast Sockets requires hardware support for Active Messages.

Project Crossbow [137] under OpenSolaris uses a hardware packet classification engine to dynamically manage priority and bandwidth assigned to a set of VMs. Its designers call this "network virtualization," but it is unrelated to network paravirtualization in PON.

The Tahoma web browsing system by Cox et al. [138] uses network proxy and a virtual

machine architecture to improve safety of access to web pages. This approach obtains many of the security advantages of PON for the specific purpose of browsing the web.

Xen, VMware, and other systems support simplified virtual Ethernet devices based on shared memory ring buffers [48, 45]. PON also uses shared memory ring buffers for virtual networking, but each of its ring buffers represents user payload data in a TCP or UDP socket, instead of a frame in a virtual Ethernet device.

7.4 File System Paravirtualization

XenFS [139] is a file system that, like POFS, shares caches between virtual machines. Unlike POFS, it is targeted specifically at Linux guests under Xen and shares Linux-specific data structures between VMs.

VNFS [71] aims to improve performance of distributed file systems where server and client run on the same host. POFS adopts both of these optimizations, as described in Section 4.3.1.

Parallax [74] demonstrates that virtual disks can be stored centrally with very high scalability. Parallax allows virtual disks to be efficiently used and modified in a copy-on-write fashion by many users. Unlike POFS, it does not allow cooperative sharing among users, nor does it enhance the transparency or improve the granularity of virtual disks.

Xen uses shared memory between VMs for access to storage [48], as does POFS. Xen, however, uses this primitive to provide access to blocks in a virtual block device, whereas POFS provides higher-level access to file data and metadata.

NFS over a remote direct memory access (RDMA) transport is reported to both increase transfer rates and reduce CPU utilization, at least for large protocol block sizes [140]. Virtual RDMA hardware in a virtual machine monitor could be used as a transport for POFS. However, RDMA is not a shared memory protocol, so such an implementation would sacrifice the cache coherency and memory savings advantages of POFS.

7.5 Virtualization Aware File Systems

VMware ESX Server includes the VMFS file system, which is designed for storing large files such as virtual disks [87]. VMFS allows for snapshots and copy-on-write sharing, but not the other features of a virtualization aware file system.

Live migration of virtual machines [141] requires the VM's storage to be available on the network. Ventana, as a distributed file system particularly suited to VM storage, provides a reasonable approach.

Whitaker et al. [142, 72] used whole-system versioning to mechanically discover the origin of a problem by doing binary search through the history of a system. They note the "semantic gap" in trying to relate changes to a virtual disk with higher-level actions. We believe that a VAFS, in which changes to files and directories may be observed directly, could help to reduce this semantic gap.

The Ventana prototype of course has much in common with other file systems. Object stores are an increasingly common way to structure file systems [83, 84, 143]. Objects in Ventana are immutable, which is unusual among object stores, although in this respect Ventana resembles the Cedar file system and, more recently, EMC's Centera system [144, 145]. PVFS2, a network file system for high-bandwidth parallel file I/O, is another file system that uses Berkeley DB databases to store file system metadata [146].

Many versioning file systems exist, in research systems such as Cedar, Elephant, and S4, and in production systems such as WAFL (used by Network Appliance filers) and VMS [144, 75, 79, 147, 148]. A versioning file system on top of a virtual disk allows old versions to be easily accessed inside the VM, but does not address the other downsides of virtual disks. None of these systems supports the tree-structured versions necessary to properly handle the natural evolution of virtual machines. The version retention policies introduced in Elephant might be usefully applied to Ventana.

Online file archives, such as Venti, also support accessing old versions of files, but again only linear versioning is supported [149].

Ventana's tree-structured version model is related to the model used in revision control systems, such as CVS [150]. A version created by merging versions from different branches has more than one parent, so versions in revision control systems are actually structured as

directed acyclic graphs. Revision control systems would generally not be good "back end" storage for Ventana or another VAFS because they typically store only a single "latest" version of a file for efficient retrieval. Retrieving other versions, including the latest version of files in branches other than the "main branch," requires application of patches [151]. Files marked "binary," however, often include each revision in full, without using patches, so use of "binary" files might be an acceptable choice.

Vesta [76] is a software configuration management system whose primary file access interface is over NFS, like Ventana. Dependency tracking in Vesta allows for precise, high-performance, repeatable builds. Similar tracking by a VAFS might enable better understanding of which files and versions should be retained over the long term.

We proposed extending a distributed file system, which already supports fine-grained sharing, by adding versioning that supports virtual machines. An alternative approach is to allow virtual disks, which already support VM-style versioning, to support sharing by adding a locking layer, as can be done for physical disks [152, 153]. This approach requires committing to a particular on-disk format, which makes changes and extensions more difficult. It also either requires each client to understand the disk format, which is a compatibility issue, or use of a network proxy that does understand the format. In the latter case the proxy is equivalent to Ventana's host manager, and the storage underlying it is really an implementation detail of the proxy.

A "union" or "overlay" file system [110, 154] can stack a writable file system above layers of read-only file systems. If the top layer is the current branch and lower layers are the branches that it was forked from, something like tree versioning can be obtained. The effect is imperfect because changes to lower layers can "show through" to the top. Symbolic link farms can also stack layers of directories, often for multi-architecture software builds [155], but link farms are not transparent to the user or software.

Application virtualization and streaming software such as SoftGrid [156] (formerly known as Softricity) offers some of the advantages of a VAFS outlined in Section 5.5, such as the ability for system administrators to install, upgrade, and patch applications on demand. Application virtualization software also offers OS-specific features not implemented in Ventana, such as the ability to virtualize the Windows registry. But application virtualization software does not allow for user customization of software, it does not enable

cooperative sharing of files among users, it does not have the flexible versioning model of a VAFS, and it does not give system administrators insight into how users are using their machines, only control over their applications.

7.6 Future Work

This thesis has demonstrated the practicality of extreme paravirtualization for two specific subsystems: networks and file systems. An obvious category of future work would be to extend extreme paravirtualization to include other domains as well. Video and sound are relatively simple candidates. More ambitious would be to attempt to move more tightly integrated kernel subsystems, such as the scheduler, into separate VMs. It is not clear that this could be done without major operating system changes.

Once major components of a kernel, such as networking and file systems, are available as separate modules, it becomes easier than currently possible to build a minimal application-specific operating system. Construction of minimal operating systems, for such purposes as reducing the trusted code base of a system of VMs (see Section 3.4), is additional future work.

Our prototype extreme paravirtualization modules were layered on top of the VMM's conventional virtual devices. Additional performance could be possible by giving the virtual device implementations direct access to the corresponding hardware.

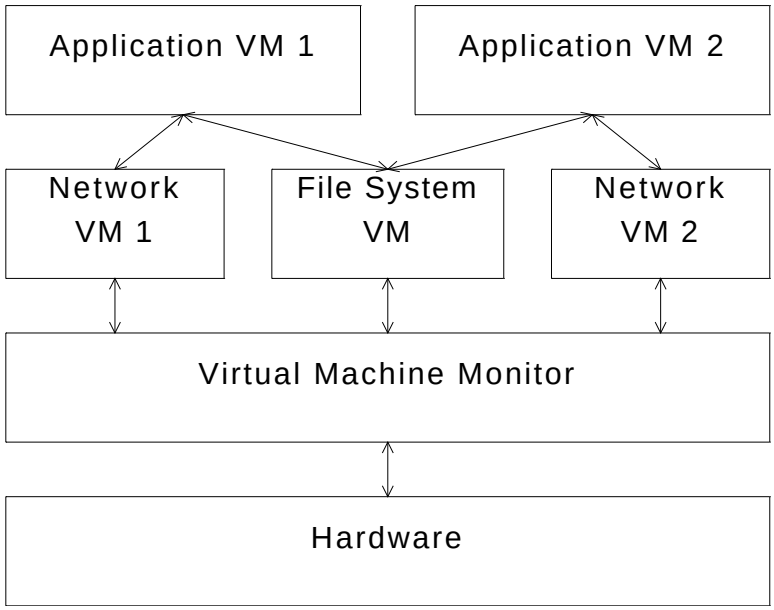

Figure 7.1: System software structure for extreme paravirtualization.

Chapter 8

Conclusion

Paravirtualizing VMMs provide virtual devices that are usually streamlined, simplified, or subsetted versions of equivalent physical hardware interfaces. This thesis investigated a more rarely seen form of paravirtual device, where the virtual interface operates at a higher level of abstraction than the common hardware interface.

We have shown the practicality of using this higher-level ("extreme") paravirtualization to better secure systems or networks, to shrink the trusted code base of an operating system, and to increase the modularity of an operating system. We demonstrated prototypes of extreme paravirtualized network and file system devices that run in VMs independent of their client VMs. We also showed that extreme paravirtualization has only a minor performance cost, even in our relatively unoptimized prototype, and that our approach offers a way to take advantage of multi-core and multi-threaded CPUs.

We further argue that extreme paravirtualization allows us to pull functionality out of modern operating system into separate, isolated environments, to form an easy path to a desirable, decomposed microkernel-like OS structure. We showed that this can be done in an evolutionary way leveraging the current industrial trends towards virtualization and multi-core CPUs, and without relying on changes in either the OS kernel or virtual machine monitor.

We further extended this extreme paravirtualization approach for virtual storage. Whereas a extreme paravirtualization interface to storage aids sharing of data on virtual disks, it does not help with the other shortcomings of virtual disks. Therefore, we proposed the concept

of a *virtualization aware file system* that combines the features of a virtual disk with those of a distributed file system. A VAFS extends a conventional distributed file system with versioning, access control, and disconnected operation features resembling those available from virtual disks. This gains the benefits of virtual disks, without compromising usability, security, or ease of management.

Bibliography

[1] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 66–77, ACM Press, 1997.

[2] J. Mersel, "Program interrupt on the Univac scientific computer," in *Western Joint Computer Conference*, pp. 52–53, American Institute of Electrical Engineers, February 1956.

[3] J. A. N. Lee, "Claims to the term "time-sharing"," *IEEE Annals of the History of Computing*, vol. 14, no. 1, pp. 16–17, 1992.

[4] J. McCarthy, "Reminiscences on the history of time-sharing," *IEEE Annals of the History of Computing*, vol. 14, no. 1, pp. 19–24, 1992. Retrieved via http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html.

[5] F. J. Corbató, M. M. Daggett, and R. C. Daley, "An experimental time-sharing system," in *Spring Joint Computer Conference*, vol. 21, American Federation of Information Processing Societies, 1962.

[6] J. McCarthy, J. Mauchly, G. Amdahl, and E. R. Piore, "Time-sharing computer systems," in *Management and the Computer of the Future* (M. Greenberger, ed.), ch. 6, pp. 220–248, M.I.T. Press, 1962.

[7] J. A. N. Lee, "Time-sharing at MIT: Introduction," *IEEE Annals of the History of Computing*, vol. 14, no. 1, pp. 13–15, 1992.

[8] R. J. Creasy, "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.

[9] J. N. Bairstow, "Many from one: the 'virtual machine' arrives," *Computer Decisions*, pp. 28–31, January 1970.

[10] L. Kleinrock, "Time-shared systems: a theoretical treatment," *Journal of the ACM*, vol. 14, no. 2, pp. 242–261, 1967.

[11] J. C. R. Licklider, "Man-computer symbiosis," *IRE Transactions on Human Factors in Electronics*, vol. 1, pp. 4–11, March 1960.

[12] W. W. Lichtenberger and M. W. Pirtle, "A facility for experimentation in man-machine interaction," in *Fall Joint Computer Conference*, vol. 27, pp. 589–598, American Federation of Information Processing Societies, 1965.

[13] A. J. Perlis, P. Elias, J. C. R. Licklider, and D. G. Marquis, "The computer in the university," in *Management and the Computer of the Future* (M. Greenberger, ed.), ch. 6, pp. 180–217, M.I.T. Press, 1962.

[14] J. A. N. Lee, "Time-sharing at MIT: The beginnings at MIT," *IEEE Annals of the History of Computing*, vol. 14, no. 1, pp. 18–30, 1992.

[15] J. W. Forgie, "A time- and memory-sharing executive program for quick-response on-line applications," in *Fall Joint Computer Conference*, vol. 27, pp. 599–609, American Federation of Information Processing Societies, 1965.

[16] M. W. Varian, "VM and the VM community: Past, present, and future," in *SHARE 89*, 1989. As revised in 1997. Available at http://www.princeton.edu/~melinda.

[17] R. W. O'Neill, "Experience using a time-shared multi-programming system with dynamic address relocation hardware," in *Spring Joint Computer Conference*, vol. 30, pp. 611–621, American Federation of Information Processing Societies (AFIPS), Thompson Books, 1967.

[18] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[19] P. J. Denning and J. Tomayko, "Origin of virtual machines and other virtualities," *IEEE Annals of the History of Computing*, vol. 23, no. 3, p. 73, 2001.

[20] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[21] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 21–36, 1964.

[22] G. E. Hoernes and L. Hellerman, "An experimental 360/40 for time-sharing," *Datamation*, vol. 14, pp. 39–42, April 1968.

[23] R. A. Meyer and L. H. Seawright, "A virtual machine time-sharing system," *IBM Systems Journal*, vol. 9, no. 3, pp. 199–218, 1970.

[24] L. Wheeler, "When was MMU virtualization first considered practical?." http://www.garlic.com/~lynn/2007i.html#14, April 2007.

[25] R. P. Parmalee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal*, vol. 7, no. 2, pp. 99–130, 1972.

[26] L. Wheeler, "Historical curiosity question." http://www.garlic.com/~lynn/2007f.html#33, March 2007.

[27] R. A. MacKinnon, "The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines," *IBM Systems Journal*, vol. 18, no. 1, pp. 18–46, 1979.

[28] IBM, Burlington, Mass., *IBM Virtual Machine Facility/370: System Programmer's Guide*, fifth ed., February 1976. Available via bitsavers.org.

[29] J. H. March, "The design and implementation of a virtual machine operating system using a virtual access method," in *ACM Workshop on Virtual Computer Systems*, (New York, NY, USA), pp. 15–18, ACM Press, 1973.

[30] R. J. Srodawa and L. A. Bates, "An efficient virtual machine implementation," in *ACM Workshop on Virtual Computer Systems*, (New York, NY, USA), pp. 43–73, ACM Press, 1973.

[31] J. P. Buzen and U. O. Gagliardi, "The evolution of virtual machine architecture," in *National Computer Conference and Exposition*, vol. 42, (New York, New York), pp. 291–299, American Federation of Information Processing Societies (AFIPS), June 1973.

[32] P. J. Denning and H. S. Stone, "An exchange of views on operating systems courses," *ACM SIGOPS Operating Systems Review*, vol. 14, no. 4, pp. 71–82, 1980.

[33] T. Richter, "virtualization in m68k microprocessors." Message-ID 5dmujcF32oecdU1@mid.dfncis.de, June 2007. Usenet article in alt.folklore.computing.

[34] Motorola Inc., *MC68020, MC68EC020 Microprocessors User's Manual*, 1st ed., 1992. As republished by FreeScale Semiconductor.

[35] Motorola Inc., *Motorola MC68030 Enhanced 32-Bit Microprocessor User's Manual*, 3rd ed., 1992. As republished by FreeScale Semiconductor.

[36] Motorola Inc., *M68040 User's Manual*, 1993.

[37] Motorola Inc., *M68060 User's Manual*, 1994.

[38] P. H. Gum, "System/370 extended architecture: Facilities for virtual machines," *IBM Journal of Research and Development*, vol. 27, no. 6, pp. 530–544, 1983.

[39] S. Nanba, N. Ohno, H. Kubo, H. Morisue, T. Ohshima, and H. Yamagishi, "VM/4: ACOS-4 virtual machine architecture," *SIGARCH Computer Architecture News*, vol. 13, no. 3, pp. 171–178, 1985.

[40] H. Umeno, K. Omachi, *et al.*, "Development of a high performance virtual machine system," in *Design and Analysis Working Group Preprints*, vol. 18-9, Information Processing Society of Japan, 1983. In Japanese. Cited in [39].

[41] K. Ono, "A method of improving the virtual machine systems," in *Design and Analysis Working Group Preprints*, vol. 18-8, Information Processing Society of Japan, 1983. In Japanese. Cited in [39].

[42] A. Vasilevsky, "The new economics of virtualization." Virtualization Blog, April 2006.

[43] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 143–156, ACM Press, 1997.

[44] S. W. Galley, "PDP-10 virtual machines," in *ACM Workshop on Virtual Computer Systems*, (New York, NY, USA), pp. 30–34, ACM Press, 1973.

[45] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor," in *USENIX Technical Conference*, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2001.

[46] P. Langdale, "vmmouse driver for Xorg is now open-source and in xorg cvs!." http://intr.overt.org/blog/?p=26, January 2006.

[47] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," in *USENIX Technical Conference*, 2002.

[48] 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 *ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 164–177, ACM Press, 2003.

[49] J. D. Neal, "VGA/SVGA video programming: Accessing the VGA display memory." http://www.osdever.net/FreeVGA/vga/vgamem.htm, 1998.

[50] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: Security challenges in virtual machine based computing environments," in *USENIX Workshop on Hot Topics in Operating Systems*, May 2005.

[51] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid, "Unix as an application program," in *USENIX Summer Technical Conference*, pp. 87–95, 1990.

[52] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier, "A new look at microkernel-based UNIX operating systems: Lessons in performance and compatibility," in *EurOpen Spring Conference*, (Tromsose, Norway), May 1991.

[53] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[54] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?," *IEEE Computer*, vol. 39, no. 5, pp. 44–51, 2006.

[55] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *USENIX Security Symposium*, 2002.

[56] G. R. Malan, D. Watson, F. Jahanian, and P. Howell, "Transport and application protocol scrubbing," in *IEEE Conference on Computer Communications*, pp. 1381–1390, 2000.

[57] U. Shankar and V. Paxson, "Active mapping: Resisting NIDS evasion without altering traffic," in *IEEE Symposium on Security and Privacy*, (Washington, DC, USA), p. 44, IEEE Computer Society, 2003.

[58] R. Clayton, "Ignoring the Great Firewall of China," in *Workshop on Privacy Enhancing Technologies*, vol. 6, (Cambridge, UK), June 2006.

[59] C. Ries, "Defeating Windows personal firewalls: Filtering methodologies, attacks, and defenses," tech. rep., VigilantMinds Inc., Pittsburgh, PA, August 2005.

[60] Microsoft Corp., "Microsoft Windows malicious software removal tool (KB890830)," August 2006.

[61] S. Pakin, V. Karamcheti, and A. Chien, "Fast messages: efficient, portable communication for workstation clusters and MPPs," *IEEE Concurrency*, vol. 5, pp. 60–72, April–June 1997.

[62] M. Ricciuti, "Microsoft plans 'hypervisor' for Longhorn." http://news.com.com/2061-10794_3-5735471.html, June 2005.

[63] VMware, Inc., "VMware Infrastructure: ESX Server." http://www.vmware.com/products/vi/esx/, September 2006.

[64] VMware, Inc., "Virtual machine communication interface." http://www.vmware.com/support/developer/vmci-sdk/, July 2007.

[65] R. M. Jensen, "A formal approach for communication between logically isolated virtual machines," *IBM Systems Journal*, vol. 18, no. 1, pp. 71–92, 1979.

[66] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000*, ch. 12–13. Microsoft Press, 3rd ed., 2000.

[67] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li, "Software support for virtual memory-mapped communication," in *International Parallel Processing Symposium*, (Washington, DC, USA), pp. 372–281, IEEE Computer Society, 1996.

[68] H. Butterworth, "Latest USB code including Xenidc documentation." Message-ID 1134751867.14087.27.camel@localhost.localdomain, December 2005. Xen development mailing list message.

[69] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes, "An implementation of the Hamlyn sender-managed interface architecture," in *USENIX/ACM SIGOPS Symposium on Operating Systems Design and Implementation*, (New York, NY, USA), pp. 245–259, ACM Press, 1996.

[70] S. R. Kleiman, "Vnodes: An architecture for multiple file system types in Sun UNIX," in *USENIX Summer Technical Conference*, pp. 238–247, 1986.

[71] X. Zhao, A. Prakash, B. Noble, and K. Borders, "Improving distributed file system performance in virtual machine environments," Tech. Rep. CSE-TR-526-06, University of Michigan, Ann Arbor, Mich., September 2006.

[72] A. Whitaker, R. S. Cox, and S. D. Gribble, "Using time travel to diagnose computer problems," in *ACM SIGOPS European Workshop*, (Leuven, Belgium), September 2004.

[73] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: Security challenges in virtual machine based computing environments," in *USENIX Workshop on Hot Topics in Operating Systems*, May 2005.

[74] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: Managing storage for a million machines," in *USENIX Workshop on Hot Topics in Operating Systems*, USENIX, May 2005.

[75] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the Elephant file system," in *ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 110–123, ACM Press, 1999.

[76] A. Heydon, R. Levin, T. Mann, and Y. Yu, "The Vesta approach to software configuration management," Research Report 168, Compaq Systems Research Center, March 2001.

[77] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 3–25, February 1992.

[78] A. Watson, P. Benn, A. G. Yoder, and H. T. Sun, "Multiprotocol data access: NFS, CIFS, and HTTP," tech. rep., Network Appliance, 2005.

[79] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger, "Self-securing storage: Protecting data in compromised systems.," in *USENIX/ACM*

SIGOPS Symposium on Operating Systems Design and Implementation, pp. 165–180, 2000.

[80] L. Huston and P. Honeyman, "Disconnected operation for AFS," in *USENIX Symposium on Mobile and Location-Independent Computing*, pp. 1–10, August 1994.

[81] J. S. Heidemann, T. W. Page, Jr., R. G. Guy, and G. J. Popek, "Primarily disconnected operation: Experiences with Ficus," in *Workshop on the Management of Replicated Data*, pp. 2–5, 1992.

[82] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2." RFC 1831, Aug. 1995.

[83] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *ACM Conference on Measurement & Modeling of Computer Systems*, (New York, NY, USA), pp. 272–284, ACM Press, 1997.

[84] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *IEEE Symposium on Mass Storage Systems and Technologies*, (Sardinia, Italy), July 2005.

[85] B. Schneier, *Applied Cryptography*. Wiley, 2nd ed., 1996.

[86] B. Callaghan, B. Pawlowski, and P. Staubach, "NFS version 3 protocol specification." RFC 1813, June 1995.

[87] "VMware ESX Server." http://www.vmware.com/products/esx, October 2005.

[88] C. Sar, P. Twohey, J. Yang, C. Cadar, and D. Engler, "Discovering malicious disks with symbolic execution," in *IEEE Symposium on Security and Privacy*, May 2006.

[89] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Network and Distributed Systems Security Symposium*, February 2003.

[90] T. Krazit, "Intel pledges 80 cores in five years." http://news.com.com/2100-1006_3-6119618.html, September 2006.

[91] P. G. Sobalvarro, *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1997.

[92] K. Mansley, "Engineering a user-level TCP for the CLAN network," in *ACM SIGCOMM Workshop on Network-I/O Convergence*, (New York, NY, USA), pp. 228–236, ACM Press, 2003.

[93] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *ACM/USENIX Conference on Virtual Execution Environments*, (New York, NY, USA), pp. 13–23, ACM Press, 2005.

[94] A. Dunkels, "The lwIP TCP/IP stack." http://www.sics.se/~adam/lwip/, March 2004.

[95] F. Bellard, "QEMU." http://fabrice.bellard.free.fr/qemu/.

[96] P. Schmidt *et al.*, "UNFS3 (user-space NFSv3 server)." http://unfs3.sourceforge.net, October 2005.

[97] VMware, Inc., "Workstation overview." http://www.vmware.com/products/ws/.

[98] Qumranet Inc., "KVM: Kernel-based virtual machine for Linux." http://kvm.qumranet.com, March 2007.

[99] Parallels, Inc., "3d graphics." http://www.parallels.com/en/products/desktop/features/3d/, July 2007.

[100] VMware, Inc., "Enabling accelerated 3-d for a virtual machine." http://www.vmware.com/support/ws5/doc/ws_vidsound_d3d_enabling_vm.html, July 2007.

[101] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *USENIX/ACM SIGOPS Symposium on Operating Systems Design and Implementation*, (San Francisco, CA), December 2004.

[102] L. Parziale, B. Frank, T. Noto, B. Robinson, and T. Russell, *Using Discontiguous Shared Segments and XIP2 Filesystems With Oracle Database 10g on Linux for IBM System z*. IBM Redbooks, IBM, June 2006.

[103] C. A. Waldspurger, "Memory resource management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[104] X. Zhao, K. Borders, and A. Prakash, "Towards protecting sensitive files in a compromised system," in *IEEE International Security in Storage Workshop*, (Washington, DC, USA), pp. 21–28, IEEE Computer Society, 2005.

[105] H. Härtig, "Security architectures revisited," in *ACM SIGOPS European Workshop*, (New York, NY, USA), pp. 16–23, ACM Press, 2002.

[106] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors," in *ACM SIGOPS European Workshop*, (New York, NY, USA), p. 22, ACM Press, 2004.

[107] A. K. Warfield, *Virtual Devices for Virtual Machines*. PhD thesis, University of Cambridge, May 2006.

[108] "Linux-VServer Project." http://linux-vserver.org/.

[109] S. Muir, L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman, "Privileged operations in the planetlab virtualised environment," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 1, pp. 75–88, 2006.

[110] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," in *Summer UKUUG Conference*, (London), pp. 1–9, July 1990.

[111] C. H. Forsyth, "The ubiquitous file server in plan 9," tech. rep., Vita Nuova Limited, York, England, June 2005.

[112] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Grible, "Constructing services with interposable virtual hardware," in *USENIX/ACM Symposium on Networked Systems Design and Implementation*, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2004.

[113] H. Raj and K. Schwan, "High performance and scalable i/o virtualization via self-virtualized devices," in *ACM SIGARCH Symposium on High Performance Distributed Computing*, (New York, NY, USA), pp. 179–188, ACM Press, 2007.

[114] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: a user-level network interface for parallel and distributed computing," in *ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 40–53, ACM Press, 1995.

[115] I. A. Pratt and K. Fraser, "Arsenic: A user-accessible gigabit ethernet interface," in *IEEE Conference on Computer Communications*, pp. 67–76, 2001.

[116] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The Collective: A cache-based system management architecture," in *USENIX/ACM Symposium on Networked Systems Design and Implementation*, (Berkeley, CA, USA), USENIX Association, 2005.

[117] M. Zhao, J. Zhang, and R. Figueiredo, "Distributed file system support for virtual machines in grid computing," in *ACM SIGARCH Symposium on High Performance Distributed Computing*, (Washington, DC, USA), pp. 202–211, IEEE Computer Society, 2004.

[118] M. Kozuch, M. Satyanarayanan, T. Bressoud, C. Helfrich, and S. Sinnamohideen, "Seamless mobile computing on fixed infrastructure," *IEEE Computer*, vol. 37, no. 7, pp. 65–72, 2004.

[119] S. Muir and J. M. Smith, "Functional divisions in the Piglet multiprocessor operating system," in *ACM SIGOPS European Workshop* (P. Guedes and J. Bacon, eds.), pp. 255–260, ACM, 1998.

[120] D. McAuley and R. Neugebauer, "A case for virtual channel processors," in *ACM SIGCOMM Workshop on Network-I/O Convergence*, (New York, NY, USA), pp. 237–242, ACM Press, 2003.

[121] G. J. Regnier, D. B. Minturn, G. L. McAlpine, V. A. Saletore, and A. Foong, "ETA: Experience with an Intel Xeon processor as a packet processing engine," *IEEE Micro*, vol. 24, no. 1, pp. 24–31, 2004.

[122] M. Rangarajan and A. Bohra, "TCP Servers: Offloading TCP processing in Internet servers," Tech. Rep. DCS-TR-481, Rutgers University, March 2002.

[123] M. A. Blumrich, R. D. Alpert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner, "Design choices in the SHRIMP system: An empirical study," in *IEEE International Symposium on Computer Architecture*, pp. 330–341, June 1998.

[124] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing network protocols at user level," *IEEE/ACM Transactions on Networking*, vol. 1, no. 5, pp. 554–565, 1993.

[125] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney, "Fast and flexible application-level networking on exokernel systems," *ACM Transactions on Computer Systems*, vol. 20, no. 1, pp. 49–83, 2002.

[126] C. Maeda and B. Bershad, "Networking performance for microkernels," in *IEEE Workshop on Workstation Operating Systems*, April 1992.

[127] Adaptec, Inc., "Adaptec TOE NAC 7711C." http://www.adaptec.com, April 2005.

[128] Alacritech, Inc., "Alacritech / technology review." http://www.alacritech.com/html/techreview.html, December 2002. Available via archive.org.

[129] B. Ang, "An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC," Tech. Rep. HPL-2001-8, HP Laboratories Palo Alto, January 2001.

[130] C. Chen and D. Griego, "Kernel korner: Improving server performance," *Linux Journal*, vol. 2002, no. 93, p. 6, 2002.

[131] Microsoft Corp., "Scalable networking: Network protocol offload—introducing TCP Chimney." http://www.microsoft.com/whdc/device/network/TCP_Chimney.mspx, April 2004.

[132] J. C. Mogul, "TCP offload is a dumb idea whose time has come.," in *USENIX Workshop on Hot Topics in Operating Systems* (M. B. Jones, ed.), pp. 25–30, USENIX, 2003.

[133] J. Garzik, "Complete TCP offload (or it's [sic] likeness)." Message-ID 3DAED8FF.2020307@pobox.com, October 2002. Linux networking mailing list message.

[134] M. Schläger, B. Rathke, A. Wolisz, and S. Bodenstein, "Advocating a remote socket architecture for Internet access using wireless LANs," *Mobile Networks and Applications*, vol. 6, no. 1, pp. 23–42, 2001.

[135] M. Kojo, K. Raatikainen, M. Liljeberg, J. Kiiskinen, and T. Alanko, "An efficient transport service for slow wireless telephone links," *IEEE Journal on Selected Areas in Communications*, vol. 15, pp. 1337–1348, September 1997.

[136] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, "High-performance local-area communication with Fast Sockets," in *USENIX Technical Conference*, pp. 257–274, 1997.

[137] Sun Microsystems, "OpenSolaris project: Crossbow: Network virtualization and resource control." http://www.opensolaris.org/os/project/crossbow/, May 2007.

[138] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A safety-oriented platform for web applications," in *IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 350–364, IEEE Computer Society, 2006.

[139] M. A. Williamson, "XenFS." http://wiki.xensource.com/xenwiki/XenFS.

[140] T. Talpey and C. Juszczak, "Nfs rdma problem statement," tech. rep., NFSv4 Working Group, July 2007.

[141] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *USENIX/ACM Symposium on Networked Systems Design and Implementation*, USENIX, 2005.

[142] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack," in *USENIX/ACM SIGOPS Symposium on Operating Systems Design and Implementation*, December 2004.

[143] Cluster File Systems, "Lustre." http://lustre.org/.

[144] D. K. Gifford, R. M. Needham, and M. D. Schroeder, "The Cedar file system," *Communications of the ACM*, vol. 31, no. 3, pp. 288–298, 1988.

[145] EMC Corporation, "EMC Centera content addressed storage system." http://www.emc.com/products/systems/centera.jsp, October 2005.

[146] "PVFS2: Parallel virtual file system 2." http://www.pvfs.org/pvfs2, October 2005.

[147] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," tech. rep., Network Appliance, 1995.

[148] K. McCoy, *VMS file system internals*. Newton, MA, USA: Digital Press, 1990.

[149] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 89–101, USENIX Association, 2002.

[150] P. Cederqvist *et al.*, *Version Management with CVS*, 2005.

[151] W. F. Tichy, "RCS—a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[152] T. McGregor and J. Cleary, "A block-based network file system," in *Australasian Computer Science Conference*, vol. 20 of *Australian Computer Science Communications*, (Perth), pp. 133–144, Springer, February 1998.

[153] R. C. Burns, *Data Management in a Distributed File System for Storage Area Networks*. PhD thesis, University of California Santa Cruz, March 2000.

[154] M. Klotzbuecher, "mini_fo: The mini fanout overlay file system." http://www.denx.de/twiki/bin/view/Know/MiniFOHome, October 2005.

[155] P. Eggert, "Multi-architecture builds using GNU `make`." http://make.paulandlesley.org/multi-arch.html, August 2000.

[156] Microsoft Corp., "Microsoft SoftGrid application virtualization." http://www.microsoft.com/systemcenter/softgrid/, September 2007.