Technical Report

UCAM-CL-TR-553
ISSN 1476-2986

Number 553

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Xen 2002

Paul R. Barham, Boris Dragovic, Keir A. Fraser,
Steven M. Hand, Timothy L. Harris,
Alex C. Ho, Evangelos Kotsovinos,
Anil V.S. Madhavapeddy, Rolf Neugebauer,
Ian A. Pratt, Andrew K. Warfield

January 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Xen 2002

The XenoServer Team
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue,
Cambridge CB3 0FD, ENGLAND
Xeno.Servers@cl.cam.ac.uk

January 9, 2003

Abstract

This report describes the design of Xen, the hypervisor developed as part of the XenoServer wide-area computing project. Xen enables the hardware resources of a machine to be virtualized and dynamically partitioned such as to allow multiple different 'guest' operating system images to be run simultaneously.

Virtualizing the machine in this manner provides flexibility, allowing different users to choose their preferred operating system (Windows, Linux, NetBSD), and also enables use of the platform as a testbed for operating systems research. Furthermore, Xen provides secure partitioning between these 'domains', and enables better resource accounting and QoS isolation than can be achieved within a conventional operating system. We show these benefits can be achieved at negligible performance cost.

We outline the design of Xen's main subsystems, and the interface exported to guest operating systems. Initial performance results are presented for our most mature guest operating system port, Linux 2.4. This report covers the initial design of Xen, leading up to our first public release which we plan to make available for download in April 2003. Further reports will update the design as our work progresses and present the implementation in more detail.

1 Introduction

The XenoServer project [6] is building a public infrastructure for wide-area distributed computing, creating a world in which XenoServer execution platforms are scattered across the globe and available for any member of the public. This allows users to run programs at points throughout the network to reduce communication latency, avoid network bottlenecks and minimize long-haul network charges. Also, it can be used to deploy large-scale experimental services, and to provide a network presence for transiently-connected mobile devices.

Our approach is distinguished from existing work on mobile agents, execution platforms, code hosting and the like by two principles:

1. **Tackling difficult problems at the same time.**
   Acceptable designs for execution environments, resource management, resource discovery, authentication, privacy, charging, billing, payment and auditing are all crucial to the success of our platform as an infrastructure service open to and accepted by the public. Existing work has tackled individual subsets of these problems, but tensions between the issues concerned mean that solutions proficient in some dimension are lacking in another.

2. **No brave new world.**
   Our platform will host applications written in today's programming languages against existing APIs – and, we believe, those written with tomorrow's languages and libraries. We do not want to mandate a particular code distribution format or a particular middleware toolkit for distributed programming.

An important component of the XenoServer project is to implement proof-of-concept systems software on which the other components, and public services, can safely be executed. This report introduces one such system; the prototype XenoServer node based on the Xen hypervisor.

The hypervisor runs directly on server hardware and dynamically partitions it between a number of *domains*, each of which hosts an instance of a *guest operating system*. A single XenoServer may host a range of Guest OSes running at the same time; it may also host several independent instances of the same Guest OS running in different domains, perhaps on behalf of different clients. The hypervisor provides just enough abstraction of the machine to allow effective isolation and resource management between these domains – the same principle pioneered in the Nemesis and Exokernel research operating systems.

The structure of this paper is as follows: Firstly, to set the general scene, we provide a high-level architecture and design which identifies the principles we are using to build the Xen-based hypervisor and prototype guest operating systems. Secondly, to aid the porting of new guest operating systems, we describe the C-language interface exported to domains running on top of the hypervisor. Finally, we describe the control interfaces that the hypervisor exports to enable higher-level management software to run as user-mode processes in a privileged control domain.

In a companion paper we introduce higher-level aspects of the system; how XenoServers advertise their facilities to clients, how clients identify XenoServers to host their tasks and how the we support a diverse range of billing and charging policies [3].

2 Overall System Architecture

Figure 1 presents the architecture of a XenoServer node based on the Xen hypervisor. As illustrated, we essentially take a virtual machine approach as pioneered by IBM VM/370 [7]. However, unlike VM/370 or more recent efforts such as VMWare [8] and Virtual PC [2], we do not attempt to completely virtualize the underlying hardware. Instead we adapt some parts of the hosted guest operating systems to work with our hypervisor; the operating system is effectively ported to a new target architecture, typically requiring changes in just the machine-dependent code. The user-level API is unchanged, thus existing binaries and operating system distributions can work unmodified.

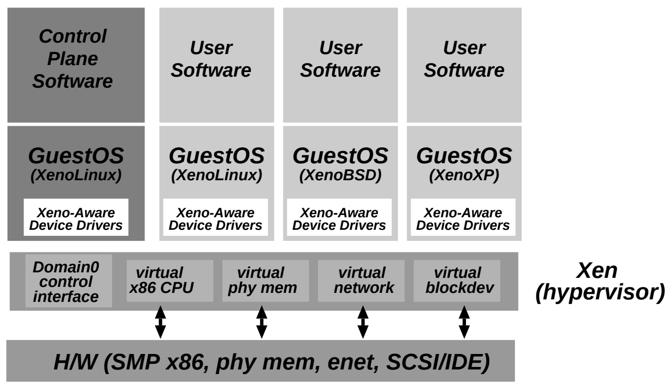

Figure 1: The structure of a XenoServer based on the Xen hypervisor, hosting a number of different guest operating systems.

This approach has been dubbed “para-virtualization” by the *Denali* project team at U. Washington [10], and provides benefits in terms of both performance and ease of implementation. This returns again to many of the observations made during the Nemesis and Exokernel projects; it is extremely difficult to fully-virtualize the existing abstractions of PC hardware. In any case, this lets Guest OSes make better informed decisions – for instance, not fully virtualizing memory lets us avoid double-paging.

As shown in the figure, the hypervisor layer serves to virtualize resources and securely multiplex access from a set of overlying virtual machines. Within the single host, there are now two levels of interface to a given resource: at the bottom level is the raw physical interface between the hypervisor and the device, and above this is the virtualized interface that is presented to the virtual machines. These interfaces, although similar, need not be identical; by making the existence of the hypervisor non-transparent it is possible to provide additional services such as scheduling and filtering to individual virtual machines.

In addition to exporting virtualized instances of the CPU, memory, network and block storage devices, Xen exposes a control interface to set how these resources are shared between the running domains. The control interface is privileged and may only by accessed by one particular virtual machine: *domain0*. This domain is a required part of any Xen-based XenoServer and runs the application software that manages the control-plane aspects of the platform – more details about these functions are available in the companion document [3].

Even in terms of a single machine, there are good reasons to keep the control software in *domain0* distinct from the hypervisor itself. In particular, it allows us to better separate the notions of *mechanism* and *policy* within the system: while the hypervisor must be involved in data-path aspects (e.g. sharing the CPU between domains, filtering network packets before transmission, or enforcing access control when reading data blocks), there is no need for it to be involved in, or even aware of, higher level issues (e.g. how the CPU is to be shared, or which kinds of packet each domain may transmit).

3 Detailed Design

In this section we introduce the design of the major subsystems that make up a Xen-based XenoServer. Rather than deal solely with the hypervisor, we present both hypervisor and Guest OS functionality for each case. This is for clarity of exposition as well as a reflecting accurately our co-design principles; the hypervisor is no use by itself.

The current discussion of Guest OSes focuses on our para-virtualized linux system, *XenoLinux*. Work is also underway building Guest OSes based on NetBSD (*XenoBSD*) and Windows XP (*XenoXP*). Reference will be made to these components where relevant.

After describing the modes of communication between the hypervisor and domains, we will tackle timers, the CPU and scheduling in Section 3.2, physical memory in Section 3.3, network devices in Section 3.4 and block devices in Section 3.5.

3.1 Interaction Between the Hypervisor and Domains

In general, domains run over the hypervisor in much the same way as processes run over an ordinary operating system; they are suspended and resumed transparently and may only invoke services from the hypervisor over a number of specified interfaces. By analogy with ordinary system calls, we term these “hypercalls” and we shall see examples of hypercall-based interfaces for various devices in the subsequent sections.

Communication from the hypervisor to a domain takes the form of “events” which stand in place of the usual delivery mechanisms for device interrupts or processor exceptions. As with traditional Unix signals, there are only a small number of events, each acting as a flag indicating a particular kind of occurrence. For instance, we shall see events used to indicate that new data has been received over the network, or that disk read requests have been completed.

As we have said, communication between a domain and the Hypervisor is similar to a standard system call interface, providing a small number of entry points for a Guest OS. However, care is taken that these calls do not block, and should be interpreted more like a notification mechanism than a full-blown system call interface. We shall see this distinction most clearly in Section 3.5 when we discuss the way in which local storage is made available to domains.

With the interfaces between the Hypervisor and the Guest OSes being only used for notifications the Hypervisor is completely asynchronous significantly simplifying its design and implementation.

Xen is entirely event-driven and requires no internal threads or processes for its own operation. It requires no dynamic memory allocation during normal operation. Some control-plane operations such as domain creation and the insertion of new network filtering rules do require new memory to be allocated, but such operations can be aborted if the memory allocation fails.

As a result of this simple structure, Xen is small, currently less than 42k lines of code excluding disk and network hardware device drivers. We hope that this will permit extensive code review and perhaps even application of formal methods to result in a stable and secure hypervisor.

3.2 Virtualizing Time and the CPU

Virtualizing the CPU involves providing abstractions for time and interrupts as well as providing a mechanism for sharing the CPU between domains. This is not very different to what a normal operating system would provide.

3.2.1 Time

Guest operating systems need to be aware of the passage of real time and their own "virtual time", i.e., the time they have been executing. Furthermore, a notion of time is required in the hypervisor itself for scheduling and the activities that relate to it. To this end, the hypervisor provides the following notions of time:

Cycle counter time provides the finest-grained, free-running time reference, with the approximate frequency being publicly accessible. The cycle counter time is used to accurately extrapolate the other time references. On SMP machines it is currently assumed that cycle counter time is synchronised between CPUs (the current x86-based implementation achieves this within inter-CPU communication latencies courtesy of the Linux initialisation code).

System time is a 64-bit value containing the nanoseconds elapsed since boot time. Unlike cycle counter time, system time accurately reflects the passage of real time, i.e., it is adjusted several times a second for timer drift. This is done by running an NTP client [4] in *domain0* on behalf of the machine, feeding updates to the hypervisor. Intermediate values can be extrapolated using the cycle counter. Providing these NTP-adjusted values through the hypervisor allows us to use a single NTP client per machine instead of performing time synchronisation in every domain.

Wall clock time is the actual "time of day" Unix style `struct timeval` (i.e., seconds and microseconds since 1 January 1970, adjusted by leap seconds etc.). Again, an NTP client hosted by *domain0* can help maintain this value. To Guest OSes this value will be reported instead of the hardware RTC clock value and they can use the system time and cycle counter times to start and remain perfectly in time.

Domain virtual time progresses at the same pace as cycle counter time, but only while a domain is executing. It stops while the domain is de-scheduled. Therefore the share of the CPU that a domain receives is indicated by the rate at which its domain virtual time increases, relative to the rate at which cycle counter time does so.

3.2.2 Timers

The hypervisor includes a timer facility which allows it to execute functions when a particular time is reached. This may be used by Xen's own device drivers and by the scheduler (see below).

The hypervisor provides Guest OSes with a limited interface to these timers. Each domain receives a pair of alarm timers; one operating in domain-virtual time and the other in system time. The Guest OSes are expected to build their own timer infrastructure (e.g. priority queue) above these. The domain virtual alarm timer can be used for scheduling within the Guest OS while all other components of the Guest OS are likely to use the system time alarm timer. We will return to the details of the timer interface for Guest OSes when describing the scheduler.

3.2.3 Virtualizing Interrupts

Interrupts are virtualized by mapping them to events, which are delivered asynchronously to the target domain. A Guest OS can map these events onto its standard interrupt dispatch mechanisms, such as using a simple vectoring scheme. Each physical interrupt source controlled by the hypervisor, including network devices, disks, or the timer subsystem, is responsible for identifying the target for an incoming interrupt and sending an event to that domain.

This demultiplexing mechanism also provides a device-specific mechanism for event coalescing or hold-off. For example, a Guest OS may request to only actually receive an event after n packets are queued ready for delivery to it, or t nanoseconds after the first packet arrived (which ever is true first). This allows us to address latency and throughput requirements on a domain-specific basis (as we did with flow-aware interrupt dispatch in the ArseNIC Gigabit Ethernet interface [5]).

3.2.4 Scheduling

Scheduling comprises two sets of interfaces. An internal interface through which a domain can notify the scheduler of its intentions and an interface through which scheduling parameters can be specified.

The internal interface to the scheduler and timers provides the following operations to each domain:

Guest OS scheduler interface
`sched_halt()`
`sched_timer(time)`
`sched_vtimer(vtime)`

Through the `sched_halt()` operation a domain relinquishes the CPU. Guest OSes are expected to call this function when idle, i.e., instead of executing an idle loop or, on x86, the `hlt` instruction. Halted domains are woken up again when receiving an event, e.g., through a virtualized interrupt caused by an arriving network packet. Domains can also request to be sent a timer event at a specific system time (using the `sched_timer()`) operation, thus defining a timeout value when halting.

While a domain is running on a CPU it can also request to receive timer events at specified domain-virtual times using the `sched_vtimer()` operation. A Guest OS can use these timer events to schedule its processes when active. The hypervisor is responsible for managing the timers operating in domain-virtual time when scheduling and de-scheduling a domain, i.e., it has to translate domain-virtual time into system time.

This very simple scheduler interface, essentially the `sched_halt()` operation in combination with event delivery and the timers operating in system time and domain-virtual time, is sufficient to build both a variety of scheduling algorithms in the hypervisor as well as supporting scheduling within Guest OSes. To improve the performance of the latter, we might in future extend the interface to include scheduler activations [1].

Underneath the scheduler interface, we are experimenting with a number of different scheduling algorithms. For example, using Nemesis' atropos scheduler to provide soft-real time absolute processor shares, and time warp to share the CPU in a weighed proportionally-fair fashion. The parameters for the scheduling algorithm are controlled via *domain0*.

There are two further aspects of the design which we must still refine in more detail, namely, how to deal with scheduling latency, especially with respect to the virtualized interrupt model described above, and improved scheduling of domains on SMP or SMT systems. Xen supports both SMT and SMP hosts, but currently each domain can run on only one processor at any time (Guest OSes are currently uni-processor). We will report on our efforts to address these issues in a future paper.

3.3 Virtualizing Physical Memory

The hypervisor is responsible for providing memory for each of the domains running over it. However, the Xen hypervisor's duty is restricted to managing physical memory and to policing page table updates. All other memory management functions are handled externally. Start-of-day issues such as building initial page tables for a domain, loading its kernel image and so on are done by the *domain builder* running in user-space within *domain0*. Paging to disk and swapping is handled by Guest OSes themselves, if they need it. This approach keeps the hypervisor lightweight while keeping the system flexible.

Many of the design decisions for the memory management subsystem were influenced by the choice of hardware architecture for the initial XenoServer platform: the *x86*. It is worth bearing in mind that the existence of certain ‘modern’ CPU features, such as a software-managed TLB, would make memory virtualisation even simpler and more efficient.

The x86 processors use a complex hybrid memory management scheme that combines paging and segmentation. Virtual addresses presented by instructions identify a segment and an offset within that segment. Segments are mapped first to a *linear address space* which is translated, via page tables, to the physical address space. Rather than just distinguishing user-mode and kernel-mode execution, there are four *rings*, ranging from 0 (the most privileged) to 3 (the least). Segments can be made accessible on a per-ring basis and a particular area in the linear address space is only accessible if it is covered by an accessible segment. Within that, pages themselves can be made accessible either to all rings, or only to rings 0-2. In an ordinary OS, each process has its own linear address space, and the OS runs in ring 0 and applications in ring 3. Segmentation, in that case, is effectively unused with each segment completely overlapping, starting at the bottom of the linear address space and extending completely to the top.

On a Xen-based XenoServer, the hypervisor itself runs in *ring 0*. It has full access to the physical memory available in the system and is responsible for allocating portions of it to the domains. Guest OSes are left to run in and use *rings 1*, *2* and *3* as they see fit, aside from the fact that segmentation is used to prevent the Guest OS from accessing a portion of the linear address space that is reserved for use by the hypervisor. This approach enables transitions between Guest OS and hypervisor without flushing the TLB, which is critical for performance. We expect most Guest OSes will use ring 1 for their own operation and place applications (if they support such a notion) in ring 3.

3.3.1 Physical Memory Allocation

The hypervisor reserves a (small) fixed portion of physical memory at system boot time. This special memory region is located at the beginning of physical memory and is mapped to the very top of every virtual address space. Except when operating in the hypervisor, i.e. in ring 0, segmentation is used to prevent access to this region. It therefore remains mapped (but not necessarily accessible) at all times – an analogous approach to how standard Linux maps kernel memory at high virtual addresses.

Any physical memory that is not used directly by the hypervisor is divided into pages and is available for allocation to domains. The hypervisor tracks which pages are free and which pages have been allocated to each domain. When a new domain is being initialized, the hypervisor allocates it pages drawn from the free list. The amount of memory required by the new domain is passed to hypervisor as one of the parameters for new domain initialization by the domain builder.

Domains can never be allocated further memory beyond that which was requested for them on initialization. However, a domain can return pages to the hypervisor if it discovers that its memory requirements have reduced. Pages might be returned to the hypervisor in several situations:

- The domain may have entered an idle phase and wishes to release unneeded resources to avoid incurring charges for their continued use.
- The domain needs a new zeroed page but the pages already allocated to it contain potentially-useful non-dirty data. Instead of swapping one of those pages out to disk, the domain can return it to the hypervisor in exchange for a fresh one.
- Once a domain has terminated all its pages are returned to the free list and are available for further allocation by the hypervisor.

The advantage of returning a page to hypervisor during execution is that the hypervisor might be able to return the very same page back to domain at a subsequent time. This is possible only in cases where the system is not under pressure for memory and thus the page in question has not been reallocated by the hypervisor between the domain returning and reclaiming it. We call this feature the *last chance cache* and *page laundering* respectively.

We are currently evaluating whether to provide read-only page sharing between domains (the

Guest OS would copy-on-write). If multiple domains are are executing or accessing the same files from a shared file system (running in *domain0*), this mechanism could provide reduced total physical memory footprint. The extent of the benefit provided by such a mechanism is under review.

3.3.2 Page Table Updates

In addition to managing physical memory allocation, the hypervisor is also in charge of performing page table updates on behalf of the domains. This is neccessary to prevent domains from adding arbitrary mappings to their page tables – for introducing mappings to one anothers' pages. Two operations are exported:

Guest OS page table update interface
`pt_set_pagetable(addr)`
`pt_update(list of requests)`

The first operation informs the hypervisor to switch the domain to use the page tables located at the specified address (of course, it must ensure suitable continuity of execution across this call). If this is a new page table, not seen before, then the hypervisor must validate the whole page table structure reachable from the specified address, and revoke any mechanisms that the domain may have to write directly to the pages containing the proposed page table (thus ensuring it remains valid).

The hypervisor allows incremental update to page tables via the `pt_update` operation. It takes a vector of proposed updates and applies them in a batch. This batching amortizes the cost of making the hypercall and exploits the fact that many Guest OS, including XenoLinux, provide explicit "flush" operations that indicate when page table updates must become visible to preserve consistency.

3.3.3 Pseudo-Physical Memory

The usual problem of external fragmentation means that a domain is unlikely to receive a contiguous stretch of physical memory. However, most Guest OSes do not have built-in support for operating in a fragmented physical address space e.g. Linux has to have a one-to-one mapping for its physical memory. Therefore a notion of *pseudo physical memory* is introduced. Once a domain is allocated a number of pages, at its start of the day, one of the first things it needs to do is to build its own *real physical* to *pseudo physical* mapping. From that moment onwards *pseudo physical* addresses are used instead of discontiguous *real physical* adresses. Thus, the rest of the Guest OS code has an impression of operating in a contiguous adress space. Guest OS page tables still contain *real* physical addresses. Mapping *pseudo physical* to *real physical* addresses is needed on page table updates and also on remapping memory regions within the Guest OS.

The decision to do the mapping inside the Guest OS and not by the hypervisor was made for two main reasons:

- To keep the hypervisor code as thin as possible, and
- Different Guest OSes may prefer the mapping done in different ways to suit their architecture. Some might not even want to do the mapping and might be happy to live in fragmented memory region.

Apart from enabling the allocation of discontiguous memory regions the *pseudo physical* approach aids domain migration – either between machines by pickling and unpickling an entire running domain, or simply between different locations within a single machine's physical address space (for example, to perform balancing within a ccNUMA machine).

3.4 Virtualizing Network Access

Since the hypervisor must multiplex network resources, its network subsystem may be viewed as a virtual network switching element with each domain having one or more virtual network interfaces attached to this network.

The conceptually simplest hypervisor design would act as a link-layer hub, forwarding all inbound traffic to all Guest OSes and relaying outbound traffic to the "real" network. In practice, however, this is undesirable due to concerns about:

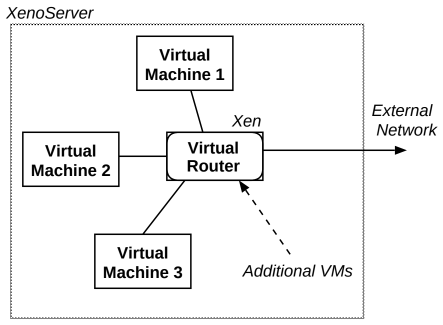

Figure 2: The Xeno Network Model

- security: any domain can snoop all traffic, and
- performance: additional copies (or complicated means of avoiding them) would be required.

Instead we choose to make the hypervisor act conceptually as an IP router, forwarding each domain's traffic according to a set of rules. The use of a general packet classifier inside the hypervisor allows additional services to be provided, as we shall see in Section 3.4.1.

As the vast majority of traffic is likely be TCP/IP-based, there is an understandable benefit in abstracting at the IP level. However, a limitation of this approach is that it does not account for non-IP (and non-ARP) traffic. Some applications which use alternative protocols might prefer a link-layer hub model which does not modify Ethernet frames before they are forwarded. If this becomes a major requirement then our design can accommodate a hybrid hub/router; however for the present we rule this out of scope. Experimental protocols may still be used via overlays (e.g. IP on IP).

3.4.1 Hypervisor Packet Handling

As described earlier, the hypervisor is responsible primarily for *data-path* operations. In terms of networking this means packet transmission and reception, which we now consider in turn.

On the transmission side, the hypervisor needs to perform two key actions:

- **Validation.** A domain is only allowed to emit packets matching a certain specification; for example, ones in which the source IP address matches one assigned to the virtual interface over which it is sent. The hypervisor is responsible for ensuring any such requirements are met, either by checking or by stamping outgoing packets with prescribed values for certain fields.
- **Scheduling.** Since a number of domains can share a single "real" network interface, the hypervisor must mediate access when several domains each have packets queued for transmission. Of course, this general scheduling function subsumes basic shaping or rate-limiting schemes.
- **Logging and Accounting.** The hypervisor can be configured with classifier rules that control how packets are accounted or logged. For example, *domain0* could request that it receives a log message or copy of the packet whenever another domain attempts to send a TCP packet containing a SYN.

On the receive side, the hypervisor's role is relatively straightforward: once a packet is received, it needs to determine the virtual interface(s) to which it must be delivered. Currently, the packet is copied to deliver it to the Guest OS. In the future, we intend to use a page-flipping scheme to eliminate the copy, trading the page containing the received packet for one from the Guest OSes free buffer ring. This should result in improved network performance at the expense of increased memory requirements due to allocating one receive buffer a page. If a smart NIC such as ArseNIC [5] is used, the packet could be delivered directly into the Guest OS receive buffer without need for copying or memory management tricks.

An appealing design approach is to abstract the demultiplexing task into a generic packet classification and trigger model. This allows a clean implementation of the receive path (see Figure 3). In addition to trivially supporting e.g. multicast, the use of other triggers allows fast forwarding, user-specified firewalling, simple packet rewriting and even clever denial-of-service (DoS) prevention techniques.

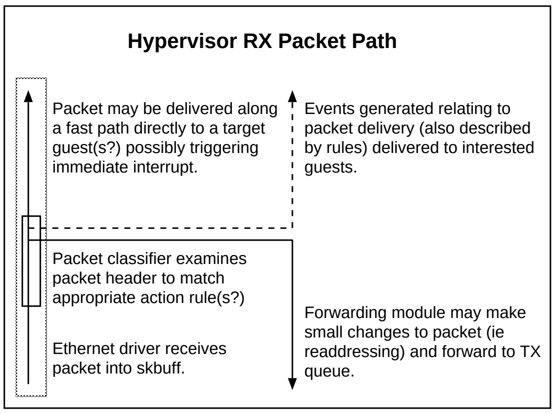
Hypervisor RX Packet Path
Packet may be delivered along a fast path directly to a target guest(s?) possibly triggering immediate interrupt.
Events generated relating to packet delivery (also described by rules) delivered to interested guests.
Packet classifier examines packet header to match appropriate action rule(s?)
Forwarding module may make small changes to packet (ie readdressing) and forward to TX queue.
Ethernet driver receives packet into skbuff.

Figure 3: Rule-Based Packet Recieve Path

For example, a set of three rules might be used to implement a simple packet symmetry scheme to prevent a domain from performing many kinds of DoS attacks against remote hosts.

- **Rule 1:** *If this inbound packet is for domain n, increment RX_count[n].*
- **Rule 2:** *If this outbound packet is from domain n, increment TX_count[n].*
- **Rule 3:** *If TX_count / RX_count > max_send_ratio, generate a symmetry violation event for domain0.*

This set of rules allows a service running on *domain0* to wait for events to arrive describing alarm conditions, such as these symmetry violations, and then take action accordingly. Updates to the classification table are initiated by passing similar event structures down from the Guest OSes. This allows a common event-based application model to easily build simple policy at the Guest OS layer.

3.4.2 Data Transfer

For network devices, actual data transfer between domains and the hypervisor uses very similar techniques to those of Nemesis or the ArseNIC gigabit Ethernet interface [5].

Unsurprisingly, the general scheme much resembles that of a user-level network interface. Each virtual interface uses two "descriptor rings", one for transmit, the other for receive. Each descriptor identifies a block of contiguous physical memory allocated to the domain. There are four cases:

- The transmit ring carries packets to transmit from the domain to the hypervisor.
- The return path of the transmit ring carries "empty" descriptors indicating that the contents have been transmitted and the memory can be re-used.
- The receive ring carries empty descriptors from the domain to the hypervisor; these provide storage space for that domain's received packets.
- The return path of the receive ring carries packets that have been received.

Real physical addresses are used throughout, with the domain performing translation from pseudo-physical addresses if that is necessary.

This scheme avoids the hypervisor from having to perform buffer management on behalf of domains – if a domain does not keep its receive ring stocked with empty buffers then packets destined to it may be dropped. It provides some defense against receive-livelock problems because an overloaded domain will cease to receive further data. Similarly, on the transmit path, it provides the application with feedback on the rate at which its packets are able to leave the system.

Synchronization between the hypervisor and the domain is achieved using counters held in shared memory that is accessible to both. Each ring has associated producer and consumer indices indicating the area in the ring that holds descriptors that contain data. After receiving n packets or t nanoseconds after receiving the first packet, the hypervisor sends an event to the domain. A single hypercall is needed to allow a domain to "kick" the hypervisor when it causes such a transition on either of the downward paths:

Guest OS network data transfer interface
`net_update()`

3.4.3 Additional Considerations

There are a number of further services that may be desirable to provide within the hypervisor:

- **Address Translation.** By performing network address translation (NAT) and port forwarding, many domains may share a common external IP address.
- **Traffic Logging.** Details regarding connections may be logged to allow forensic auditing in the case of a specific domain acting maliciously.
- **Packet Sniffing.** In order to perform network analysis, the classifier can be configure with rules to cause certain packets to be duplicated and delivered to multiple domains.

Further discussion about these and other potential features may be found in our design note [9].

3.5 Access to Block Devices

As we introduced in Figure 1, the hypervisor implements the actual device drivers for specific pieces of hardware and then exposes virtual interfaces to Guest OSes. This model is followed for IDE and SCSI drives, as it was for network devices.

As before, however, we are faced with a number of choices in terms of the level at which the underlying block device is exposed to particular domains and the kinds of sharing which we permit to occur. For instance, should the interface to local storage operate at the level of file accesses within a particular kind of file system? Alternatively, should the hypervisor provide "raw" disk access to complete physical disks?

3.5.1 Control Interfaces

The approach we have taken is a pragmatic balance between these extremes; the hypervisor provides domains with the abstraction of logical block devices built from extents drawn from the physical disks fitted to the system. Each extent in the list is given as a tuple specifying a physical drive, offset within it, extent-length and a flag indicating whether read/write or read-only access is to be granted. A control interface is provided to *domain0* to configure logical block devices on behalf of domains; the hypervisor is only responsible for the data-path protection and translation of requests:

Domain0 block device interface

```
bd_device_update (dom, devn, extent_list)
```

Domain0 has complete read/write access to their entire contents, including partition tables, boot information and so on. It will typically be configured to use one of the partitions as its root file system. Other partitions may be marked as type "XenoDisk". These will be divided up into fixed-length extents (currently 100MB chunks), and available for *domain0* to construct logical block devices from, as necessary.

In a typical installation, *domain0* will use some extents to hold a shared file system that it exports to other domains via a local NFS or SMB server. This provides domains with a high-level file-based read/write interface. In the future, a shared memory block transfer mechanism will augment the NFS server for improved efficiency when servicing Guest OS file requests. The memory management system will thus enable pages that are in use by more than one domain to be shared read-only (copy-on-write) between domains.

Rather than only exporting a local file system via this mechanism, we expect *domain0* will wish to provide domains with access to other distributed file systems, such as Coda, Tapestry, Ivy, or our own XenoStore. In which case, it will likely use some of the local disk extents to provide a persistent shared cache of recently accessed files. With the shared-memory block transport in place, this could provide an efficient means of allowing Guest OSes to retrieve files that are likely to exhibit a high degree of sharing, such as common system and application binaries.

Instead of using the file-level interface, some domains will choose to ask *domain0* to construct a logical block device for them. For example, a domain might ask that a new block device be created such that it may perform paging (swaping) to it. Doing so will provide the domain with high-performance disk access, bypassing the file-level shared cache which could provide no benefit anyway.

Similarly, a domain could request creation of a new block device, build a file system on it, and populate it with important application data that it needs to access with predictable performance, and thus could not rely on the distributed file system.

3.5.2 Persistent block devices

When created, block devices are reserved for the client's exclusive use during the period the Guest OS is running on the server. However, we wish to provide the option of persistent storage, allowing a client to reclaim a block device previously created by another Guest OS.

Upon creation, logical block device are assigned a pair of large sparse identifiers which serve as access keys. One identifier provides read-write access, the other read-only. When asking *domain0* to construct it a logical block device, a domain in possession of a key can use it to request a specific pre-existing block device containing known data, perhaps encrypted for additional security. This mechanism also allows Guest OS to perform read-only sharing of common data, perhaps root file system images.

When created, a persistent block device can be assigned an expiry date. If not actively renewed by someone in possession of the key, the disk extents that make up the block device are liable to be zeroed and re-allocated on an LRU basis as new block devices are created or existing ones extended.

3.5.3 Data Transfer

Domains which have been granted access to a logical block device are permitted to read and write it directly through the hypervisor, rather than requiring *domain0* to mediate every data access.

In overview, we use the same style of descriptor-ring that is used for network packets. Each domain has one ring that carries operation requests to the hypervisor and carries the results back again. Three kinds of operation request are possible. Although the actual communication uses an asynchronous message passing model, they are most easily understood in terms of a procedural interface in that manner that they could be exposed by an abstraction library *within the domain*. The first three procedures place an operation on the descriptor ring and return a handle for that invocation:

Guest OS block device request messages

```
handle = msg_bd_read(device, buffer,
                     blk_num, blk_cnt)
handle = msg_bd_write(device, buffer,
                     blk_num, blk_cnt)
handle = msg_bd_barrier(device)
```

The first two of these operations, `msg_bd_read` and `msg_bd_write` request ordinary asynchronous read and write operations on the specified logical block device and at the specified range of real physical memory locations. Remember that these procedures are not hypercalls; they operate entirely within a domain, placing messages into a request-descriptor ring shared with the hypervisor.

These operations permit contiguous transfers between block devices and guest operating systems. With these primitives, it is also possible to support more advanced operations such as scatter-gather operations.

The third operation, `msg_bd_barrier`, is worthy of more detailed attention. We permit the hypervisor to re-order operations on block devices, allowing it to perform any of the set of operations that have been requested of it by any of the domains. The `msg_bd_barrier` request acts as a synchronization barrier that a domain can use to constrain the re-ordering. It requires that the hypervisor completes servicing all of the operations that precede the barrier before commencing any of the operations that follow it. This supports domain-specific trade-offs in the consistency model they require.

The final three messages form counterparts of those above and indicate to the guest operating system the status of the request identified by the handle:

Guest OS block device response messages

```
msg_bd_read_reply(handle)
msg_bd_write_reply(handle)
msg_bd_barrier_reply(handle)
```

As we have seen, the majority of block-device operations are carried by these requests and responses over a descriptor ring. Aside from these sets of messages, only a single hypercall is necessary. Once again, this is used to inform the hypervisor when it should check for new data placed within the descriptor ring:

Guest OS block device data transfer interface
`bd_update()`

3.5.4 DMA Transfers

Rather than copying data in and out of the hypervisor, we use page pinning to enable DMA transfers directly between the physical device and the domain's buffers. Disk read operations are straightforward; the hypervisor just needs to know which pages have pending DMA transfers, and prevent the Guest OS from trying to give them page back to the hypervisor, or to use them for storing page tables. Similarly, if a page is to be subject of a disk write operation, it must not be reallocated, otherwise the new owner's contents could be leaked. Note that unlike the case of network transmission, we do not have to be concerned about the security implications of a domain updating a page's contents between initiating a write and it being performed.

4 Preliminary Results

We have plans to perform a thorough analysis of Xen's performance, comparing Guest OSes running over Xen against the same OS running on the bare hardware. We will use a variety of different work loads, including SPEC CPU, SPEC WEB (using Apache), and micro-benchmarks such as lmbench. For comparison, we also intend to repeat the experiments with Linux running over VMWare, preferably VMware ESX server. We hope to show that the performance overhead of para-virtualization is negligible, and certainly less that the full virtualization aimed for by VMWare.

The most mature Guest OS port we currently have running on Xen is Linux 2.4. Table 4 show results comparing the performance of the SPEC CPU INT 2000 suite running on xenolinux-2.4.20 on Xen *vs.* Linux-2.4.20 running on the same hardware (a dual processor 733MHz Pentium-III, 512MB). The main purpose of this benchmark is to thoroughly exercise Xen's memory virtualization technique. We arranged for the benchmark suite to be running in an NFS-mounted directory, thus giving Xen's network stack a work-out too. The results are very encouraging – running within Xenolinux, the benchmark's overall performance degradation is less than 2%. We also have initial results from running the lmbench microbenchmark suite. Although these present an absolute worse-case scenario, most are in the 10-20% overhead range. Given that no concerted effort has yet been made to optimize either Xen or Xenolinux, we find these figures very encouraging. We will run experiments to provide further performance results shortly.

5 Conclusion

We have presented the design of the Xen Hypervisor and XenoLinux, a guest OS which runs on top of it. The project is ongoing and under active development by an enthusiastic team. We will continue to improve the implementation and to evolve our design, with the aim of having our system selected for deployment on the PlanetLab test bed.

6 Credits

The XenoServer hypervisor team are (in alphabetical order): Paul Barham, Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Evangelos Kotsovinos, Anil Madhavapeddy, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Keir Fraser deserves special mention for leading the initial implementation.

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM*

Table 1: SPEC CPU INT2000 suite running on a 733Mhz dual Pentium-III system. XenoLinux *vs.* Linux

	XenoLinux 2.4		Linux 2.4	
benchmark	Run Time	Ratio	Run Time	Ratio
164.gzip	487	287	482	291
175.vpr	640	219	623	225
176.gcc	475	232	454	242
181.mcf	930	194	881	204
186.crafty	316	316	315	317
197.parser	762	236	748	241
252.eon	1824	71.3	1833	70.9
253.perlbmk	530	340	530	340
254.gap	415	265	403	273
255.vortex	640	297	630	302
256.bzip2	692	217	676	222
300.twolf	1236	243	1212	248
SPECint_base2000		229.4		234.0

SIGOPS Symposium on Operating Systems Principles (SOSP'91) (Pacific Grove, CA, USA, Oct. 1991), pp. 95–109.

[2] CORP., C. The technology of Virtual PC, 2000. Available from http://www.connectix.com/downloadcenter /pdf/vpcw_wp/vpcw_overviewwp _sep1301.pdf.

[3] HAND, S., HARRIS, T., KOTSOVINOS, E., AND PRATT, I. Controlling the XenoServer Open Platform, November 2002. Under submission to OpenArch'03.

[4] MILLS, D. L. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, University of Delaware, Mar. 1993.

[5] PRATT, I., AND FRASER, K. Arsenic: A User-Accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)* (Los Alamitos, CA, Apr. 22–26 2001), IEEE Computer Society, pp. 67–76.

[6] REED, D., PRATT, I., MENAGE, P., EARLY, S., AND STRATFORD, N. Xenoservers: accounted execution of untrusted code.

[7] SEAWRIGHT, L., AND MACKINNON, R. VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems Journal* (1979), 4–17.

[8] VMware virtual platform, technical white paper, 1999.

[9] WARFIELD, A., HAND, S., HARRIS, T., AND PRATT, I. Isolation of Network Resources in XenoSevers, November 2002. Planetlab Design Note PDN-02-006.

[10] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Tech. Rep. Technical Report 02-02-01, University of Washington, 2002.