Xen and the Art of Open Source Virtualization

Keir Fraser, Steven Hand, Christian Limpach, Ian Pratt
University of Cambridge Computer Laboratory
`{first.last}@cl.cam.ac.uk`

Dan Magenheimer
Hewlett-Packard Laboratories
`{first.last}@hp.com`

Abstract

Virtual machine (VM) technology has been around for 40 years and has been experiencing a resurgence with commodity machines. VMs have been shown to improve system and network flexibility, availability, and security in a variety of novel ways. This paper introduces Xen, an efficient secure open source VM monitor, to the Linux community.

Key features of Xen are:

1. supports different OSes (e.g. Linux 2.4, 2.6, NetBSD, FreeBSD, etc.)
2. provides secure protection between VMs
3. allows flexible partitioning of resources between VMs (CPU, memory, network bandwidth, disk space, and bandwidth)
4. very low overhead, even for demanding server applications
5. support for seamless, low-latency migration of running VMs within a cluster

We discuss the interface that Xen/x86 exports to guest operating systems, and the kernel changes that were required to Linux to port it to Xen. We compare Xen/Linux to User Mode Linux as well as existing commercial VM products.

1 Introduction

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller virtual machines (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation, co-located hosting facilities, distributed web services, secure computing platforms, and application mobility.

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our Xen-Linux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.6. Xen ports of NetBSD and FreeBSD have been completed, along with a proof of concept port of Windows XP.[1]

There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes—protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of computational grids, or the experimental PlanetLab platform [11]), but not when resources are oversubscribed, or users uncooperative.

One way to address this problem is to retrofit support for performance isolation to the operating system, but a difficulty with such approaches is ensuring that *all* resource usage is accounted to the correct process—consider, for example, the complex interactions between applications due to buffer cache or page replacement algorithms. Performing multiplexing at a low level can mitigate this problem; unintentional or undesired interactions between tasks are minimized. Xen multiplexes physical resources at the granularity of an entire operating system and is able to provide performance isolation between them. This allows a range of guest operating systems to gracefully coexist rather than mandating a specific application binary interface. There is a price to pay for this flexibility—running a full OS is more heavyweight than running a process, both in terms of initialization (e.g. booting or resuming an OS instance versus `fork`/`exec`), and in terms of resource consumption.

For our target of 10-100 hosted OS instances, we believe this price is worth paying: It allows individual users to run unmodified binaries, or collections of binaries, in a resource controlled fashion (for instance an Apache server along with a PostgreSQL backend). Furthermore it provides an extremely high level of flexibility since the user can dynamically create the precise execution environment their software requires. Unfortunate configuration interactions between various services and applications are avoided (for example, each Windows instance maintains its own registry).

Experience with deployed Xen systems suggests that the initialization overheads and additional resource requirements are in practice quite low: An operating system image may be resumed from an on-disk snapshot in typically just over a second (depending on image memory size), and although multiple copies of the operating system code and data are stored in memory, the memory requirements are typically small compared to those of the applications that will run on them. As we shall show later in the paper, the performance overhead of the virtualization provided by Xen is low, typically just a few percent, even for the most demanding applications.

[1]The Windows XP port required access to Microsoft source code, and hence distribution is currently restricted, even in binary form.

2 XEN: Approach & Overview

In a traditional VMM the virtual hardware exposed is functionally identical to the underlying machine [14]. Although *full virtualization* has the obvious benefit of allowing unmodified operating systems to be hosted, it also has a number of drawbacks. This is particularly true for the prevalent Intel *x86* architecture.

Support for full virtualization was never part of the x86 architectural design. Certain supervisor instructions must be handled by the VMM for correct virtualization, but executing these with insufficient privilege fails silently rather than causing a convenient trap [13]. Efficiently virtualizing the x86 MMU is also difficult. These problems can be solved, but only at the cost of increased complexity and reduced performance. VMware's ESX Server [3] dynamically rewrites portions of the hosted machine code to insert traps wherever VMM intervention might be required. This translation is applied to the entire guest OS kernel (with associated translation, execution, and caching costs) since all non-trapping privileged instructions must be caught and handled. ESX Server implements shadow versions of system structures such as page tables and maintains consistency with the virtual tables by trapping every update attempt—this approach has a high cost for update-intensive operations such as creating a new application process.

Notwithstanding the intricacies of the x86, there are other arguments against full virtualization. In particular, there are situations in which it is desirable for the hosted operating systems to see real as well as virtual resources: providing both real and virtual time allows a guest OS to better support time-sensitive tasks, and to correctly handle TCP timeouts and RTT estimates, while exposing real machine addresses allows a guest OS to improve performance by using superpages [10] or page coloring [7].

We avoid the drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware—an approach which has been dubbed *paravirtualization* [17]. This promises improved performance, although it does require modifications to the guest operating system. It is important to note, however, that we do not require changes to the application binary interface (ABI), and hence no modifications are required to guest *applications*.

We distill the discussion so far into a set of design principles:

1. Support for unmodified application binaries is essential, or users will not transition to Xen. Hence we must virtualize all architectural features required by existing standard ABIs.

2. Supporting full multi-application operating systems is important, as this allows complex server configurations to be virtualized within a single guest OS instance.

3. Paravirtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as x86.

4. Even on cooperative machine architectures, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

In the following section we describe the virtual machine abstraction exported by Xen and discuss how a guest OS must be modified to conform to this. Note that in this paper we reserve the term *guest operating system* to refer to one of the OSes that Xen can host and we use the term *domain* to refer to a running virtual machine within which a guest OS executes; the distinction is analogous to that between a *program* and a *process* in a conventional system. We call Xen itself the *hypervisor* since it operates at a higher privilege level than the supervisor code of the guest operating systems that it hosts.

2.1 The Virtual Machine Interface

The paravirtualized x86 interface can be factored into three broad aspects of the system: memory management, the CPU, and device I/O. In the following we address each machine subsystem in turn, and discuss how each is presented in our paravirtualized architecture. Note that although certain parts of our implementation, such as memory management, are specific to the x86, many aspects (such as our virtual CPU and I/O devices) can be readily applied to other machine architectures. Furthermore, x86 represents a *worst case* in the areas where it differs significantly from RISC-style processors—for example, efficiently virtualizing hardware page tables is more difficult than virtualizing a software-managed TLB.

2.1.1 Memory management

Virtualizing memory is undoubtedly the most difficult part of paravirtualizing an architecture, both in terms of the mechanisms required in the hypervisor and modifications required to port each guest OS. The task is easier if the architecture provides a software-managed TLB as these can be efficiently virtualized in a simple manner [5]. A tagged TLB is another useful feature supported by most server-class RISC architectures, including Alpha, MIPS and SPARC. Associating an address-space identifier tag with each TLB entry allows the hypervisor and each guest OS to efficiently coexist in separate address spaces because there is no need to flush the entire TLB when transferring execution.

Unfortunately, x86 does not have a software-managed TLB; instead TLB misses are serviced automatically by the processor by walking the page table structure in hardware. Thus to achieve the best possible performance, all valid page translations for the current address space should be present in the hardware-accessible page table. Moreover, because the TLB is not tagged, address space switches typically require a complete TLB flush. Given these limitations, we made two decisions: (i) guest OSes are responsible for allocating and managing the hardware page tables, with minimal involvement from Xen to ensure safety and isolation; and (ii) Xen exists in a 64MB section at the top of every address space, thus avoiding a TLB flush when entering and leaving the hypervisor.

Each time a guest OS requires a new page table, perhaps because a new process is being created, it allocates and initializes a page from its own memory reservation and registers it with Xen. At this point the OS must relinquish direct write privileges to the page-table memory: all subsequent updates must be validated by Xen. This restricts updates in a number of ways, including only allowing an OS to map pages that it owns, and disallowing writable mappings of page tables. Guest OSes may *batch* update requests to amortize the overhead of entering the hypervisor. The top 64MB region of each address space, which is reserved for Xen, is not accessible or remappable by guest OSes. This address region is not used by any of the common x86 ABIs however, so this restriction does not break application compatibility.

Segmentation is virtualized in a similar way, by validating updates to hardware segment descriptor tables. The only restrictions on x86 segment descriptors are: (i) they must have lower privilege than Xen, and (ii) they may not allow any access to the Xen-reserved portion of the address space.

2.1.2 CPU

Virtualizing the CPU has several implications for guest OSes. Principally, the insertion of a hypervisor below the operating system violates the usual assumption that the OS is the most privileged entity in the system. In order to protect the hypervisor from OS misbehavior (and domains from one another) guest OSes must be modified to run at a lower privilege level.

Efficient virtualizion of privilege levels is possible on x86 because it supports four distinct privilege levels in hardware. The x86 privilege levels are generally described as *rings*, and are numbered from zero (most privileged) to three (least privileged). OS code typically executes in ring 0 because no other ring can execute privileged instructions, while ring 3 is generally used for application code. To our knowledge, rings 1 and 2 have not been used by any well-known x86 OS since OS/2. Any OS which follows this common arrangement can be ported to Xen by modifying it to execute in ring 1. This prevents the guest OS from directly executing privileged instructions, yet it remains safely isolated from applications running in ring 3.

Privileged instructions are paravirtualized by requiring them to be validated and executed within Xen—this applies to operations such as installing a new page table, or yielding the processor when idle (rather than attempting to `hlt` it). Any guest OS attempt to directly execute a privileged instruction is failed by the processor, either silently or by taking a fault, since only Xen executes at a sufficiently privileged level.

Exceptions, including memory faults and software traps, are virtualized on x86 very straightforwardly. A table describing the handler for each type of exception is registered with Xen for validation. The handlers specified in this table are generally identical to those for real x86 hardware; this is possible because the exception stack frames are unmodified in our paravirtualized architecture. The sole modification is to the page fault handler, which would normally read the faulting address from a privileged processor register (`CR2`); since this is not possible, we write it into an extended stack frame[2]. When an exception occurs while executing outside ring 0, Xen's handler creates a copy of the exception stack frame on the guest OS stack and returns control to the appropriate registered handler.

Typically only two types of exception occur frequently enough to affect system performance: system calls (which are usually implemented via a software exception), and page faults. We improve the performance of system calls by allowing each guest OS to register a 'fast' exception handler which is accessed directly by the processor without indirecting via ring 0; this handler is validated before installing it in the hardware exception table. Unfortunately it is not possible to apply the same technique to the page fault handler because only code executing in ring 0 can read the faulting address from register `CR2`; page faults must therefore always be delivered via Xen so that this register value can be saved for access in ring 1.

Safety is ensured by validating exception handlers when they are presented to Xen. The only required check is that the handler's code segment does not specify execution in ring 0. Since no guest OS can create such a segment,

[2]In hindsight, writing the value into a pre-agreed shared memory location rather than modifying the stack frame would have simplified the XP port.

it suffices to compare the specified segment selector to a small number of static values which are reserved by Xen. Apart from this, any other handler problems are fixed up during exception propagation—for example, if the handler's code segment is not present or if the handler is not paged into memory then an appropriate fault will be taken when Xen executes the `iret` instruction which returns to the handler. Xen detects these "double faults" by checking the faulting program counter value: if the address resides within the exception-virtualizing code then the offending guest OS is terminated.

Note that this "lazy" checking is safe even for the direct system-call handler: access faults will occur when the CPU attempts to directly jump to the guest OS handler. In this case the faulting address will be outside Xen (since Xen will never execute a guest OS system call) and so the fault is virtualized in the normal way. If propagation of the fault causes a further "double fault" then the guest OS is terminated as described above.

2.1.3 Device I/O

Rather than emulating existing hardware devices, as is typically done in fully-virtualized environments, Xen exposes a set of clean and simple device abstractions. This allows us to design an interface that is both efficient and satisfies our requirements for protection and isolation. To this end, I/O data is transferred to and from each domain via Xen, using shared-memory, asynchronous buffer-descriptor rings. These provide a high-performance communication mechanism for passing buffer information vertically through the system, while allowing Xen to efficiently perform validation checks (for example, checking that buffers are contained within a domain's memory reservation).

Linux subsection	# lines
Architecture-independent	78
Virtual network driver	484
Virtual block-device driver	1070
Xen-specific (non-driver)	1363
Total	**2995**
Portion of total x86 code base	**1.36%**

Table 1: The simplicity of porting commodity OSes to Xen.

Similar to hardware interrupts, Xen supports a lightweight event-delivery mechanism which is used for sending asynchronous notifications to a domain. These notifications are made by updating a bitmap of pending event types and, optionally, by calling an event handler specified by the guest OS. These callbacks can be 'held off' at the discretion of the guest OS—to avoid extra costs incurred by frequent wake-up notifications, for example.

2.2 The Cost of Porting an OS to Xen

Table 1 demonstrates the cost, in lines of code, of porting commodity operating systems to Xen's paravirtualized x86 environment.

The architecture-specific sections are effectively a port of the x86 code to our paravirtualized architecture. This involved rewriting routines which used privileged instructions, and removing a large amount of low-level system initialization code.

2.3 Control and Management

Throughout the design and implementation of Xen, a goal has been to separate policy from mechanism wherever possible. Although the hypervisor must be involved in data-path aspects (for example, scheduling the CPU between domains, filtering network packets before transmission, or enforcing access control

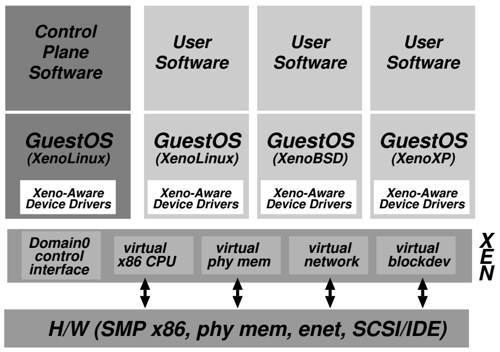

Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenLinux environment.

when reading data blocks), there is no need for it to be involved in, or even aware of, higher level issues such as how the CPU is to be shared, or which kinds of packet each domain may transmit.

The resulting architecture is one in which the hypervisor itself provides only basic control operations. These are exported through an interface accessible from authorized domains; potentially complex policy decisions, such as admission control, are best performed by management software running over a guest OS rather than in privileged hypervisor code.

The overall system structure is illustrated in Figure 1. Note that a domain is created at boot time which is permitted to use the *control interface*. This initial domain, termed *Domain0*, is responsible for hosting the application-level management software. The control interface provides the ability to create and terminate other domains and to control their associated scheduling parameters, physical memory allocations and the access they are given to the machine's physical disks and network devices.

In addition to processor and memory resources, the control interface supports the creation and deletion of virtual network interfaces (VIFs) and block devices (VBDs). These virtual I/O devices have associated access-control information which determines which domains can access them, and with what restrictions (for example, a read-only VBD may be created, or a VIF may filter IP packets to prevent source-address spoofing or apply traffic shaping).

This control interface, together with profiling statistics on the current state of the system, is exported to a suite of application-level management software running in *Domain0*. This complement of administrative tools allows convenient management of the entire server: current tools can create and destroy domains, set network filters and routing rules, monitor per-domain network activity at packet and flow granularity, and create and delete virtual network interfaces and virtual block devices.

Snapshots of a domains' state may be captured and saved to disk, enabling rapid deployment of applications by bypassing the normal boot delay. Further, Xen supports *live migration* which enables running VMs to be moved dynamically between different Xen servers, with execution interrupted only for a few milliseconds. We are in the process of developing higher-level tools to further automate the application of administrative policy, for example, load balancing VMs among a cluster of Xen servers.

3 Detailed Design

In this section we introduce the design of the major subsystems that make up a Xen-based server. In each case we present both Xen and guest OS functionality for clarity of exposition. In this paper, we focus on the XenLinux guest OS; the *BSD and Windows XP ports use the Xen interface in a similar manner.

3.1 Control Transfer: Hypercalls and Events

Two mechanisms exist for control interactions between Xen and an overlying domain: synchronous calls from a domain to Xen may be made using a *hypercall*, while notifications are delivered to domains from Xen using an asynchronous event mechanism.

The hypercall interface allows domains to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page-table updates, in which Xen validates and applies a list of updates, returning control to the calling domain when this is completed.

Communication from Xen to a domain is provided through an asynchronous event mechanism, which replaces the usual delivery mechanisms for device interrupts and allows lightweight notification of important events such as domain-termination requests. Akin to traditional Unix signals, there are only a small number of events, each acting to flag a particular type of occurrence. For instance, events are used to indicate that new data has been received over the network, or that a virtual disk request has completed.

Pending events are stored in a per-domain bitmask which is updated by Xen before invoking an event-callback handler specified by the guest OS. The callback handler is responsible for resetting the set of pending events, and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen-readable software flag: this is analogous to disabling interrupts on a real processor.

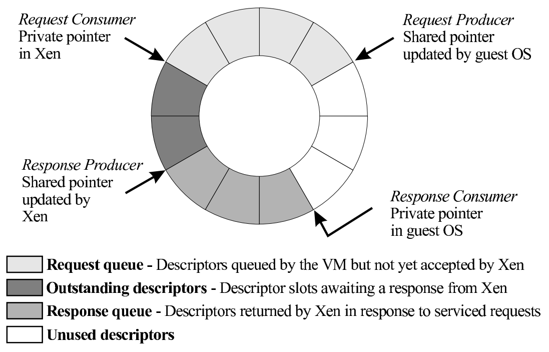
Request Consumer
Private pointer in Xen
Request Producer
Shared pointer updated by guest OS
Response Producer
Shared pointer updated by Xen
Response Consumer
Private pointer in guest OS
Request queue - Descriptors queued by the VM but not yet accepted by Xen
Outstanding descriptors - Descriptor slots awaiting a response from Xen
Response queue - Descriptors returned by Xen in response to serviced requests
Unused descriptors

Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

3.2 Data Transfer: I/O Rings

The presence of a hypervisor means there is an additional protection domain between guest OSes and I/O devices, so it is crucial that a data transfer mechanism be provided that allows data to move vertically through the system with as little overhead as possible.

Two main factors have shaped the design of our I/O-transfer mechanism: resource management and event notification. For resource accountability, we attempt to minimize the work required to demultiplex data to a specific domain when an interrupt is received from a device—the overhead of managing buffers is carried out later where computation may be accounted to the appropriate domain. Similarly, memory committed to device I/O is provided by the relevant domains wherever possible to prevent the crosstalk inherent in shared buffer pools; I/O buffers are protected during data transfer by pinning the underlying page frames within Xen.

Figure 2 shows the structure of our I/O descriptor rings. A ring is a circular queue of descriptors allocated by a domain but accessible from within Xen. Descriptors do not directly contain I/O data; instead, I/O data buffers are allocated out-of-band by the guest OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly, save with Xen as the producer and the guest OS as the consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations.

This structure is sufficiently generic to support a number of different device paradigms. For example, a set of 'requests' can provide buffers for network packet reception; subsequent 'responses' then signal the arrival of packets into these buffers. Reordering is useful when dealing with disk requests as it allows them to be scheduled within Xen for efficiency, and the use of descriptors with out-of-band buffers makes implementing zero-copy transfer easy.

We decouple the production of requests or responses from the notification of the other party: in the case of requests, a domain may enqueue multiple entries before invoking a hypercall to alert Xen; in the case of responses, a domain can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to trade-off latency and throughput requirements, similarly to the flow-aware interrupt dispatch in the ArseNIC Gigabit Ethernet interface [12].

3.3 Subsystem Virtualization

The control and data transfer mechanisms described are used in our virtualization of the various subsystems. In the following, we discuss how this virtualization is achieved for CPU, timers, memory, network and disk.

3.3.1 CPU scheduling

Xen currently schedules domains according to the Borrowed Virtual Time (BVT) scheduling algorithm [4]. We chose this particular algorithms since it is both work-conserving and has a special mechanism for low-latency wake-up (or *dispatch*) of a domain when it receives an event. Fast dispatch is particularly important to minimize the effect of virtualization on OS subsystems that are designed to run in a timely fashion; for example, TCP relies on the timely delivery of acknowledgments to correctly estimate network round-trip times. BVT provides low-latency dispatch by using virtual-time warping, a mechanism which temporarily violates 'ideal' fair sharing to favor recently-woken domains. However, other scheduling algorithms could be trivially implemented over our generic scheduler abstraction. Per-domain scheduling parameters can be adjusted by management software running in *Domain0*.

3.3.2 Time and timers

Xen provides guest OSes with notions of real time, virtual time and wall-clock time. Real time is expressed in nanoseconds passed since machine boot and is maintained to the accuracy of the processor's cycle counter and can be frequency-locked to an external time source (for example, via NTP). A domain's virtual time only advances while it is executing: this is typically used by the guest OS scheduler to ensure correct sharing of its timeslice between application processes. Finally, wall-clock time is specified as an offset to be added to the current real time. This allows the wall-clock time to be adjusted without affecting the forward progress of real time.

Each guest OS can program a pair of alarm timers, one for real time and the other for virtual time. Guest OSes are expected to maintain internal timer queues and use the Xen-provided alarm timers to trigger the earliest timeout. Timeouts are delivered using Xen's event mechanism.

3.3.3 Virtual address translation

As with other subsystems, Xen attempts to virtualize memory access with as little overhead as possible. As discussed in Section 2.1.1, this goal is made somewhat more difficult by the x86 architecture's use of hardware page tables. The approach taken by VMware is to provide each guest OS with a virtual page table, not visible to the memory-management unit (MMU) [3]. The hypervisor is then responsible for trapping accesses to the virtual page table, validating updates, and propagating changes back and forth between it and the MMU-visible 'shadow' page table. This greatly increases the cost of certain guest OS operations, such as creating new virtual address spaces, and requires explicit propagation of hardware updates to 'accessed' and 'dirty' bits.

Although full virtualization forces the use of shadow page tables, to give the illusion of contiguous physical memory, Xen is not so constrained. Indeed, Xen need only be involved in page table *updates*, to prevent guest OSes from making unacceptable changes. Thus we avoid the overhead and additional complexity associated with the use of shadow page tables—the approach in Xen is to register guest OS page tables directly with the MMU, and restrict guest OSes to read-only access. Page table updates are passed to Xen via a hypercall; to ensure safety, requests are *validated* before being applied.

To aid validation, we associate a type and reference count with each machine page frame. A frame may have any one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings to its own page frames, regardless of their current types. A frame may only safely be retasked when its reference count is zero. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the frame concerned to simultaneously be of types PT and RW.

The type system is also used to track which frames have already been validated for use in page tables. To this end, guest OSes indicate when a frame is allocated for page-table use—this requires a one-off validation of every entry in the frame by Xen, after which its type is pinned to PD or PT as appropriate, until a subsequent unpin request from the guest OS. This is particularly useful when changing the page table base pointer, as it obviates the need to validate the new page table on every context switch. Note that a frame cannot be retasked until it is both unpinned and its reference count has reduced to zero – this prevents guest OSes from using unpin requests to circumvent the reference-counting mechanism.

3.3.4 Physical memory

The initial memory allocation, or *reservation*, for each domain is specified at the time of its creation; memory is thus statically partitioned between domains, providing strong isolation. A maximum-allowable reservation may also be specified: if memory pressure within a domain increases, it may then attempt to claim additional memory pages from Xen, up to this reservation limit. Conversely, if a domain wishes to save resources, perhaps to avoid incurring unnecessary costs, it can reduce its memory reservation by releasing memory pages back to Xen.

XenLinux implements a *balloon driver* [16], which adjusts a domain's memory usage by passing memory pages back and forth between Xen and XenLinux's page allocator. Although we could modify Linux's memory-management routines directly, the balloon driver makes adjustments by using existing OS functions, thus simplifying the Linux porting effort. However, paravirtualization can be used to extend the capabilities of the balloon driver; for example, the out-of-memory handling mechanism in the guest OS can be modified to automatically alleviate memory pressure by requesting more memory from Xen.

Most operating systems assume that memory comprises at most a few large contiguous extents. Because Xen does not guarantee to allocate contiguous regions of memory, guest OSes will typically create for themselves the illusion of contiguous *physical memory*, even though their underlying allocation of *hardware memory* is sparse. Mapping from physical to hardware addresses is entirely the responsibility of the guest OS, which can simply maintain an array indexed by physical page frame number. Xen supports efficient hardware-to-physical mapping by providing a shared translation array that is directly readable by all domains – updates to this array are validated by Xen to ensure that the OS concerned owns the relevant hardware page frames.

Note that even if a guest OS chooses to ignore hardware addresses in most cases, it must use the translation tables when accessing its page tables (which necessarily use hardware addresses). Hardware addresses may also be exposed to limited parts of the OS's memory-management system to optimize memory access. For example, a guest OS might allocate particular hardware pages so as to optimize placement within a physically indexed cache [7], or map naturally aligned contiguous portions of hardware memory using superpages [10].

3.3.5 Network

Xen provides the abstraction of a virtual firewall-router (VFR), where each domain has one or more network interfaces (VIFs) logically attached to the VFR. A VIF looks somewhat like a modern network interface card: there are two I/O rings of buffer descriptors, one for transmit and one for receive. Each direction also has a list of associated rules of the form ($<$*pattern*$>$, $<$*action*$>$)—if the *pattern* matches then the associated *action* is applied.

Domain0 is responsible for inserting and removing rules. In typical cases, rules will be installed to prevent IP source address spoofing, and to ensure correct demultiplexing based on destination IP address and port. Rules may also be associated with hardware interfaces on the VFR. In particular, we may install rules to perform traditional firewalling functions such as preventing incoming connection attempts on insecure ports.

To transmit a packet, the guest OS simply enqueues a buffer descriptor onto the transmit ring. Xen copies the descriptor and, to ensure safety, then copies the packet header and executes any matching filter rules. The packet payload is not copied since we use scatter-gather DMA; however note that the relevant page frames must be pinned until transmission is complete. To ensure fairness, Xen implements a simple round-robin packet scheduler.

To efficiently implement packet reception, we require the guest OS to exchange an unused page frame for each packet it receives; this avoids the need to copy the packet between Xen and the guest OS, although it requires that page-aligned receive buffers be queued at the network interface. When a packet is received, Xen immediately checks the set of receive rules to determine the destination VIF, and exchanges the packet buffer for a page frame on the relevant receive ring. If no frame is available, the packet is dropped.

3.3.6 Disk

Only *Domain0* has direct unchecked access to physical (IDE and SCSI) disks. All other domains access persistent storage through the abstraction of virtual block devices (VBDs), which are created and configured by management software running within *Domain0*. Allowing *Domain0* to manage the VBDs keeps the mechanisms within Xen very simple and avoids more intricate solutions such as the UDFs used by the Exokernel [6].

A VBD comprises a list of extents with associated ownership and access control information, and is accessed via the I/O ring mechanism. A typical guest OS disk scheduling algorithm will reorder requests prior to enqueuing them on the ring in an attempt to reduce response time, and to apply differentiated service (for example, it may choose to aggressively schedule synchronous metadata requests at the expense of speculative readahead requests). However, because Xen has more complete knowledge of the actual disk layout, we also support reordering within Xen, and so responses may be returned out of order. A VBD thus appears to the guest OS somewhat like a SCSI disk.

A translation table is maintained within the hypervisor for each VBD; the entries within this table are installed and managed by *Domain0* via a privileged control interface. On receiving a disk request, Xen inspects the VBD identifier and offset and produces the corresponding sector address and physical device. Permission checks also take place at this time. Zero-copy data transfer takes place using DMA between the disk and pinned memory pages in the requesting domain.

Xen services *batches* of requests from competing domains in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disk hardware. Domains may explicitly pass down *reorder barriers* to prevent reordering when this is necessary to maintain higher level semantics (e.g. when using a write-ahead log). The low-level scheduling gives us good throughput, while the batching of requests provides reasonably fair access. Future work will investigate providing more predictable isolation and differentiated service, perhaps using existing techniques and schedulers [15].

4 Evaluation

In this section we present a subset of our evaluation of Xen against a number of alternative virtualization techniques. A more complete evaluation, as well as detailed configuration and benchmark specs, can be found in [1] For these measurements, we used our 2.4.21-based XenLinux port as, at the time of this writing, the 2.6-port was not stable enough for a full battery of tests.

There are a number of preexisting solutions for running multiple copies of Linux on the same machine. VMware offers several commercial products that provide virtual x86 machines on which unmodified copies of Linux may be booted. The most commonly used version is VMware Workstation, which consists of a set of privileged kernel extensions to a 'host' operating system. Both Windows and Linux hosts are supported. VMware also offer an enhanced product called ESX Server which replaces the host OS with a dedicated kernel. By doing so, it gains some performance benefit over the workstation product. We have subjected ESX Server to the benchmark suites described below, but sadly are prevented from reporting quantitative results due to the terms of the product's End User License Agreement. Instead we present results from VMware Workstation 3.2, running on top of a Linux host OS, as it is the most recent VMware product without that benchmark publication restriction. ESX Server takes advantage of its native architecture to equal or outperform VMware Workstation and its hosted architecture. While Xen of course requires guest OSes to be ported, it takes advantage of paravirtualization to noticeably outperform ESX Server.

We also present results for User-mode Linux (UML), an increasingly popular platform for virtual hosting. UML is a port of Linux to run as a user-space process on a Linux host. Like XenLinux, the changes required are restricted to the architecture dependent code base. However, the UML code bears little similarity to the native x86 port due to the very different nature of the execution environments. Although UML can run on an unmodified Linux host, we present results for the 'Single Kernel Address Space' (skas3) variant that exploits patches to the host OS to improve performance.

We also investigated three other virtualization techniques for running ported versions of Linux on the same x86 machine. Connectix's Virtual PC and forthcoming Virtual Server products (now acquired by Microsoft) are similar in design to VMware's, providing full x86 virtualization. Since all versions of Virtual PC have benchmarking restrictions in their license agreements we did not subject them to closer analysis. UMLinux is similar in concept to UML but is a different code base and has yet to achieve the same level of performance, so we omit the results. Work to improve the performance of UMLinux through host OS modifications is ongoing [8]. Although Plex86 was originally a general purpose x86 VMM, it has now been retargeted to support just Linux guest OSes. The guest OS must be specially compiled to run on Plex86, but the source changes from native x86 are trivial. The performance of Plex86 is currently well below the other techniques.

4.1 Relative Performance

The first cluster of bars in Figure 3 represents a relatively easy scenario for the VMMs. The SPEC CPU suite contains a series of long-running computationally-intensive applications intended to measure the performance of a system's processor, memory system, and compiler quality. The suite performs little I/O and has little interaction with the OS. With almost all CPU time spent executing in user-space code, all three VMMs exhibit low overhead.

The next set of bars show the total elapsed time taken to build a default configuration of the Linux 2.4.21 kernel on a local ext3 file system with gcc 2.96. Native Linux spends about 7% of the CPU time in the OS, mainly performing file I/O, scheduling and memory management. In the case of the VMMs, this 'system time' is expanded to a greater or lesser degree: whereas Xen incurs a mere 3% overhead, the other VMMs experience a more significant slowdown.

Two experiments were performed using the PostgreSQL 7.1.3 database, exercised by the Open Source Database Benchmark suite (OSDB) in its default configuration. We present results for the multi-user Information

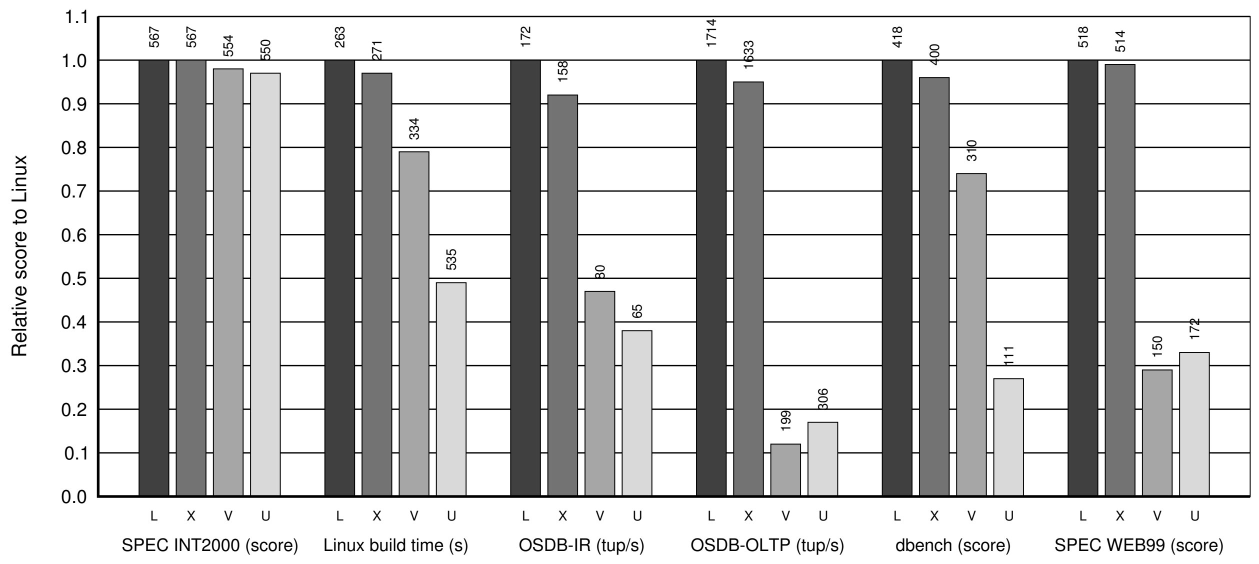

Figure 3: Relative performance of native Linux (L), XenLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

Retrieval (IR) and On-Line Transaction Processing (OLTP) workloads, both measured in tuples per second. PostgreSQL places considerable load on the operating system, and this is reflected in the substantial virtualization overheads experienced by VMware and UML. In particular, the OLTP benchmark requires many synchronous disk operations, resulting in many protection domain transitions.

The `dbench` program is a file system benchmark derived from the industry-standard 'NetBench'. It emulates the load placed on a file server by Windows 95 clients. Here, we examine the throughput experienced by a single client performing around 90,000 file system operations.

SPEC WEB99 is a complex application-level benchmark for evaluating web servers and the systems that host them. The benchmark is CPU-bound, and a significant proportion of the time is spent within the guest OS kernel, performing network stack processing, file system operations, and scheduling between the many `httpd` processes that Apache needs to handle the offered load. XenLinux fares well, achieving within 1% of native Linux performance. VMware and UML both struggle, supporting less than a third of the number of clients of the native Linux system.

4.2 Operating System Benchmarks

To more precisely measure the areas of overhead within Xen and the other VMMs, we performed a number of smaller experiments targeting particular subsystems. We examined the overhead of virtualization as measured by McVoy's *lmbench* program [9]. The OS performance subset of the lmbench suite consist of 37 microbenchmarks.

In 24 of the 37 microbenchmarks, XenLinux performs similarly to native Linux, tracking the Linux kernel performance closely. In Tables 2 to 4 we show results which exhibit interesting performance variations among the test systems; particularly large penalties for Xen are shown in bold face.

In the process microbenchmarks (Table 2), Xen

Config	null call	null I/O	open close	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
Linux	0.45	0.50	1.92	5.70	0.68	2.49	110	530	4k0
Xen	0.46	0.50	1.88	5.69	0.69	1.75	**198**	**768**	**4k8**
VMW	0.73	0.83	2.99	11.1	1.02	4.63	874	2k3	10k
UML	24.7	25.1	62.8	39.9	26.0	46.0	21k	33k	58k

Table 2: `lmbench`: Processes - times in μs

Config	2p 0K	2p 16K	2p 64K	8p 16K	8p 64K	16p 16K	16p 64K
Linux	0.77	0.91	1.06	1.03	24.3	3.61	37.6
Xen	**1.97**	**2.22**	**2.67**	**3.07**	**28.7**	**7.08**	39.4
VMW	18.1	17.6	21.3	22.4	51.6	41.7	72.2
UML	15.5	14.6	14.4	16.3	36.8	23.6	52.0

Table 3: `lmbench`: Context switching times in μs

Config	0K File create	0K File delete	10K File create	10K File delete	Mmap lat	Prot fault	Page fault
Linux	32.1	6.08	66.0	12.5	68.0	1.06	1.42
Xen	32.5	5.86	68.2	13.6	**139**	1.40	**2.73**
VMW	35.3	9.3	85.6	21.4	620	7.53	12.4
UML	130	65.7	250	113	1k4	21.8	26.3

Table 4: `lmbench`: File & VM system latencies in μs

exhibits slower *fork*, *exec*, and *sh* performance than native Linux. This is expected, since these operations require large numbers of page table updates which must all be verified by Xen. However, the paravirtualization approach allows XenLinux to batch update requests. Creating new page tables presents an ideal case: because there is no reason to commit pending updates sooner, XenLinux can amortize each hypercall across 2048 updates (the maximum size of its batch buffer). Hence each update hypercall constructs 8MB of address space.

Table 3 shows context switch times between different numbers of processes with different working set sizes. Xen incurs an extra overhead between 1μs and 3μs, as it executes a hypercall to change the page table base. However, context switch results for larger working set sizes (perhaps more representative of real applications) show that the overhead is small compared with cache effects. Unusually, VMware Workstation is inferior to UML on these microbenchmarks; however, this is one area where enhancements in ESX Server are able to reduce the overhead.

The *mmap latency* and *page fault latency* results shown in Table 4 are interesting since they require two transitions into Xen per page: one to take the hardware fault and pass the details to the guest OS, and a second to install the updated page table entry on the guest OS's behalf. Despite this, the overhead is relatively modest.

One small anomaly in Table 2 is that XenLinux has lower signal-handling latency than native Linux. This benchmark does not require any calls into Xen at all, and the 0.75μs (30%) speedup is presumably due to a fortuitous cache alignment in XenLinux, hence underlining the dangers of taking microbenchmarks too seriously.

4.3 Additional Benchmarks

We have also conducted comprehensive experiments that: evaluate the overhead of virtualizing the network; compare the performance of running multiple applications in their own guest OS against running them on the same native operating system; demonstrate performance isolation provided by Xen; and examine Xen's ability to scale to its target of 100 domains. All of the experiments showed promising results and details have been separately published [1].

5 Conclusion

We have presented the Xen hypervisor which partitions the resources of a computer between domains running guest operating systems. Our paravirtualizing design places a particular emphasis on protection, performance and resource management. We have also described and evaluated XenLinux, a fully-featured port of the Linux kernel that runs over Xen.

Xen and the 2.4-based XenLinux are sufficiently stable to be useful to a wide audience. Indeed, some web hosting providers are already selling Xen-based virtual servers. Sources, documentation, and a demo ISO can be found on our project page[3].

Although the 2.4-based XenLinux was the basis of our performance evaluation, a 2.6-based port is well underway. In this port, much care is been given to minimizing and isolating the necessary changes to the Linux kernel and measuring the changes against benchmark results. As paravirtualization techniques become more prevalent, kernel changes would ideally be part of the main tree. We have experimented with various source structures including a separate architecture, *a la* UML, a subarchitecture, and a CONFIG option. We eagerly solicit input and discussion from the kernel developers to guide our approach. We also have considered transparent paravirtualization [2] techniques to allow a single distro image to adapt dynamically between a VMM-based configuration and bare metal.

As well as further guest OS ports, Xen itself is being ported to other architectures. An x86_64 port is well underway, and we are keen to see Xen ported to RISC-style architectures (such as PPC) where virtual memory virtualization will be much easier due to the software-managed TLB.

Much new functionality has been added since the first public availability of Xen last October. Of particular note are a completely revamped I/O subsystem capable of directly utilizing Linux driver source, suspend/resume and live migration features, much improved console access, etc. Though final implementation, testing, and documentation was not complete at the deadline for this paper, we hope to describe these in more detail at the symposium and in future publications.

[3]`http://www.cl.cam.ac.uk/netos/xen`

As always, there are more tasks to do than there are resources to do them. We would like to grow Xen into the premier open source virtualization solution, with breadth and features that rival proprietary commercial products.

We enthusiastically welcome the help and contributions of the Linux community.

Acknowledgments

Xen has been a big team effort. In particular, we would like to thank Zachary Amsden, Andrew Chung, Richard Coles, Boris Dragovich, Evangelos Kotsovinos, Tim Harris, Alex Ho, Kip Macy, Rolf Neugebauer, Bin Ren, Russ Ross, James Scott, Steven Smith, Andrew Warfield, Mark Williamson, and Mike Wray. Work on Xen has been supported by a UK EPSRC grant, Intel Research, Microsoft Research, and Hewlett-Packard Labs.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles*, volume 37(5) of *ACM Operating Systems Review*, pages 164–177, Bolton Landing, NY, USA, Dec. 2003.

[2] D. Magenheimer and T. Christian. vBlades: Optimized Paravirtualization for the Itanium Processor Family. In *Proceedings of the*

USENIX 3rd Virtual Machine Research and Technology Symposium, May 2004.

[3] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent*, 6397242, Oct. 1998.

[4] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 261–276, Kiawah Island Resort, SC, USA, Dec. 1999.

[5] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-level virtual memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[6] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 52–65, Oct. 1997.

[7] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Transaction on Computer Systems*, 10(4):338–359, Nov. 1992.

[8] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 Annual USENIX Technical Conference*, Jun 2003.

[9] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, Berkeley, Jan. 1996. Usenix Association.

[10] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 89–104, Boston, MA, USA, Dec. 2002.

[11] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, NJ, USA, Oct. 2002.

[12] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, Los Alamitos, CA, USA, Apr. 22–26 2001. IEEE Computer Society.

[13] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA*, pages 129–144, Aug. 2000.

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

[15] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of ACM SIGMETRICS'98, the International Conference on Measurement and Modeling of Computer Systems*, pages 44–55, June 1998.

[16] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 181–194, Boston, MA, USA, Dec. 2002.

[17] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada