Xen and the Art of Virtualization

Paul Barham*, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris,
Alex Ho, Rolf Neugebauer†, Ian Pratt, Andrew Warfield

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK, CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service.

This paper presents Xen, an x86 virtual machine monitor which allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacrificing either performance or functionality. This is achieved by providing an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP, can be *ported* with minimal effort.

Our design is targeted at hosting up to 100 virtual machine instances simultaneously on a modern server. The virtualization approach taken by Xen is extremely efficient: we allow operating systems such as Linux and Windows XP to be hosted simultaneously for a negligible performance overhead — at most a few percent compared with the unvirtualized case. We considerably outperform competing commercial and freely available solutions in a range of microbenchmarks and system-wide tests.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Measurement, Performance

Keywords

Virtual Machine Monitors, Hypervisors, Paravirtualization

*Microsoft Research Cambridge, UK
†Intel Research Cambridge, UK

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

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 [42, 8], co-located hosting facilities [14], distributed web services [43], secure computing platforms [12, 16] and application mobility [26, 37].

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 XenoLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.4. Our port of Windows XP to Xen is not yet complete but is capable of running simple user-space processes. Work is also progressing in porting NetBSD.

Xen enables users to dynamically instantiate an operating system to execute whatever they desire. In the XenoServer project [15, 35] we are deploying Xen on standard server hardware at economically strategic locations within ISPs or at Internet exchanges. We perform admission control when starting new virtual machines and expect each VM to *pay* in some fashion for the resources it requires. We discuss our ideas and approach in this direction elsewhere [21]; this paper focuses on the VMM.

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 com-

putational grids, or the experimental PlanetLab platform [33]), 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. This has been demonstrated to a greater or lesser degree with resource containers [3], Linux/RK [32], QLinux [40] and SILK [4]. One 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. This is effectively the problem of "QoS crosstalk" [41] within the operating system. Performing multiplexing at a low level can mitigate this problem, as demonstrated by the Exokernel [23] and Nemesis [27] operating systems. Unintentional or undesired interactions between tasks are minimized.

We use this same basic approach to build Xen, which multiplexes physical resources at the granularity of an entire operating system and is able to provide performance isolation between them. In contrast to process-level multiplexing this also 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 versus `fork` and `exec`), and in terms of resource consumption.

For our target of up to 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).

The remainder of this paper is structured as follows: in Section 2 we explain our approach towards virtualization and outline how Xen works. Section 3 describes key aspects of our design and implementation. Section 4 uses industry standard benchmarks to evaluate the performance of XenoLinux running above Xen in comparison with stand-alone Linux, VMware Workstation and User-mode Linux (UML). Section 5 reviews related work, and finally Section 6 discusses future work and concludes.

2. XEN: APPROACH & OVERVIEW

In a traditional VMM the virtual hardware exposed is functionally identical to the underlying machine [38]. 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 IA-32, or *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 [36]. 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 [10] 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 [30] or page coloring [24].

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* [43]. 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.

Note that our paravirtualized x86 abstraction is quite different from that proposed by the recent Denali project [44]. Denali is designed to support thousands of virtual machines running network services, the vast majority of which are small-scale and unpopular. In contrast, Xen is intended to scale to approximately 100 virtual machines running industry standard applications and services. Given these very different goals, it is instructive to contrast Denali's design choices with our own principles.

Firstly, Denali does not target existing ABIs, and so can elide certain architectural features from their VM interface. For example, Denali does not fully support x86 segmentation although it is exported (and widely used[1]) in the ABIs of NetBSD, Linux, and Windows XP.

Secondly, the Denali implementation does not address the problem of supporting application multiplexing, nor multiple address spaces, within a single guest OS. Rather, applications are linked explicitly against an instance of the Ilwaco guest OS in a manner rather reminiscent of a libOS in the Exokernel [23]. Hence each virtual machine essentially hosts a single-user single-application unprotected "operating system". In Xen, by contrast, a single virtual machine hosts a real operating system which may itself securely multiplex thousands of unmodified user-level processes. Although a prototype *virtual MMU* has been developed which may help Denali in this area [44], we are unaware of any published technical details or evaluation.

Thirdly, in the Denali architecture the VMM performs all paging to and from disk. This is perhaps related to the lack of memory-management support at the virtualization layer. Paging within the

[1]For example, segments are frequently used by thread libraries to address thread-local data.

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

Table 1: The paravirtualized x86 interface.

VMM is contrary to our goal of performance isolation: malicious virtual machines can encourage thrashing behaviour, unfairly depriving others of CPU time and disk bandwidth. In Xen we expect each guest OS to perform its own paging using its own guaranteed memory reservation and disk allocation (an idea previously exploited by self-paging [20]).

Finally, Denali virtualizes the 'namespaces' of all machine resources, taking the view that no VM can access the resource allocations of another VM if it cannot name them (for example, VMs have no knowledge of hardware addresses, only the virtual addresses created for them by Denali). In contrast, we believe that secure access control within the hypervisor is sufficient to ensure protection; furthermore, as discussed previously, there are strong correctness and performance arguments for making physical resources directly visible to guest OSes.

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

Table 1 presents an overview of the paravirtualized x86 interface, 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 [13]. 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.

Many processor architectures only provide two privilege levels. In these cases the guest OS would share the lower privilege level with applications. The guest OS would then protect itself by running in a separate address space from its applications, and indirectly pass control to and from applications via the hypervisor to set the virtual privilege level and change the current address space. Again, if the processor's TLB supports address-space tags then expensive TLB flushes can be avoided.

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

OS subsection	# lines	
	Linux	**XP**
Architecture-independent	78	1299
Virtual network driver	484	–
Virtual block-device driver	1070	–
Xen-specific (non-driver)	1363	3321
Total	**2995**	**4620**
(**Portion of total x86 code base**	**1.36%**	**0.04%**)

Table 2: The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).

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).

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 2 demonstrates the cost, in lines of code, of porting commodity operating systems to Xen's paravirtualized x86 environment. Note that our NetBSD port is at a very early stage, and hence we report no figures here. The XP port is more advanced, but still in progress; it can execute a number of user-space applications from a RAM disk, but it currently lacks any virtual I/O drivers. For this reason, figures for XP's virtual device drivers are not presented. However, as with Linux, we expect these drivers to be small and simple due to the idealized hardware abstraction presented by Xen.

Windows XP required a surprising number of modifications to its architecture independent OS code because it uses a variety of structures and unions for accessing page-table entries (PTEs). Each page-table access had to be separately modified, although some of

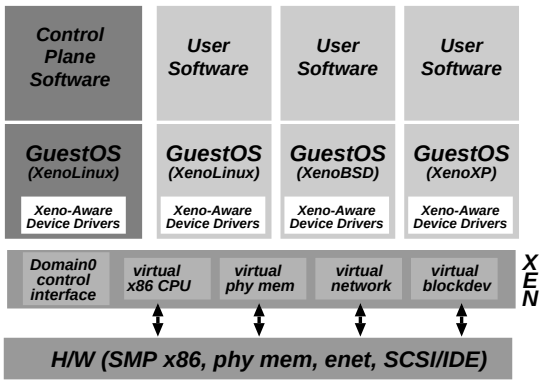

Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenoLinux environment.

this process was automated with scripts. In contrast, Linux needed far fewer modifications to its generic memory system as it uses preprocessor macros to access PTEs — the macro definitions provide a convenient place to add the translation and hypervisor calls required by paravirtualization.

In both OSes, 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. Again, more changes were required in Windows XP, mainly due to the presence of legacy 16-bit emulation code and the need for a somewhat different boot-loading mechanism. Note that the x86-specific code base in XP is substantially larger than in Linux and hence a larger porting effort should be expected.

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 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).

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. We anticipate the development of higher-level tools to further automate the application of administrative policy.

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. The current discussion of guest OSes focuses on XenoLinux as this is the most mature; nonetheless our ongoing porting of Windows XP and NetBSD gives us confidence that Xen is guest OS agnostic.

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.

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.

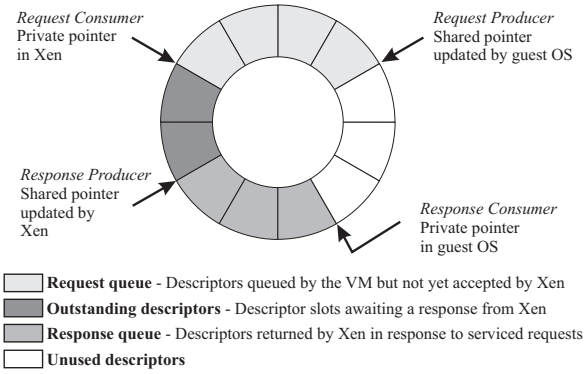

Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

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 [34].

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 [11]. 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) [10]. 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.

To minimize the number of hypercalls required, guest OSes can locally queue updates before applying an entire batch with a single hypercall — this is particularly beneficial when creating new address spaces. However we must ensure that updates are committed early enough to guarantee correctness. Fortunately, a guest OS will typically execute a TLB flush before the first use of a new mapping: this ensures that any cached translation is invalidated. Hence, committing pending updates immediately before a TLB flush usually suffices for correctness. However, some guest OSes elide the flush when it is certain that no stale entry exists in the TLB. In this case it is possible that the first attempted use of the new mapping will cause a page-not-present fault. Hence the guest OS fault handler must check for outstanding updates; if any are found then they are flushed and the faulting instruction is retried.

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.

XenoLinux implements a *balloon driver* [42], which adjusts a domain's memory usage by passing memory pages back and forth between Xen and XenoLinux'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 [24], or map naturally aligned contiguous portions of hardware memory using superpages [30].

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 [23].

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 us-

ing 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 [39].

3.4 Building a New Domain

The task of building the initial guest OS structures for a new domain is mostly delegated to *Domain0* which uses its privileged control interfaces (Section 2.3) to access the new domain's memory and inform Xen of initial register state. This approach has a number of advantages compared with building a domain entirely within Xen, including reduced hypervisor complexity and improved robustness (accesses to the privileged interface are sanity checked which allowed us to catch many bugs during initial development).

Most important, however, is the ease with which the building process can be extended and specialized to cope with new guest OSes. For example, the boot-time address space assumed by the Linux kernel is considerably simpler than that expected by Windows XP. It would be possible to specify a fixed initial memory layout for all guest OSes, but this would require additional bootstrap code within every guest OS to lay things out as required by the rest of the OS. Unfortunately this type of code is tricky to implement correctly; for simplicity and robustness it is therefore better to implement it within *Domain0* which can provide much richer diagnostics and debugging support than a bootstrap environment.

4. EVALUATION

In this section we present a thorough performance evaluation of Xen. We begin by benchmarking Xen against a number of alternative virtualization techniques, then compare the total system throughput executing multiple applications concurrently on a single native operating system against running each application in its own virtual machine. We then evaluate the performance isolation Xen provides between guest OSes, and assess the total overhead of running large numbers of operating systems on the same hardware. For these measurements, we have used our XenoLinux port (based on Linux 2.4.21) as this is our most mature guest OS. We expect the relative overheads for our Windows XP and NetBSD ports to be similar but have yet to conduct a full evaluation.

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. ESX Server also supports a paravirtualized interface to the network that can be accessed by installing a special device driver (vmxnet) into the guest OS, where deployment circumstances permit.

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. We are authorized simply to say that "Xen significantly outperforms ESX Server" on all benchmarks in the suite. We present results for VMware Workstation 3.2, the most recent VMware product released without the benchmarking restriction, running over a Linux host OS.

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 XenoLinux, 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 [25]. 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.

All the experiments were performed on a Dell 2650 dual processor 2.4GHz Xeon server with 2GB RAM, a Broadcom Tigon 3 Gigabit Ethernet NIC, and a single Hitachi DK32EJ 146GB 10k RPM SCSI disk. Linux version 2.4.21 was used throughout, compiled for architecture *i686* for the native and VMware guest OS experiments, for *xeno-i686* when running on Xen, and architecture *um* when running on UML. The Xeon processors in the machine support SMT ("hyperthreading"), but this was disabled because none of the kernels currently have SMT-aware schedulers. We ensured that the total amount of memory available to all guest OSes plus their VMM was equal to the total amount available to native Linux.

The RedHat 7.2 distribution was used throughout, installed on ext3 file systems. The VMs were configured to use the same disk partitions in 'persistent raw mode', which yielded the best performance. Using the same file system image also eliminated potential differences in disk seek times and transfer rates.

4.1 Relative Performance

We have performed a battery of experiments in order to evaluate the overhead of the various virtualization techniques relative to running on the 'bare metal'. Complex application-level benchmarks that exercise the whole system have been employed to characterize performance under a range of server-type workloads. Since neither Xen nor any of the VMware products currently support multiprocessor guest OSes (although they are themselves both SMP capable), the test machine was configured with one CPU for these experiments; we examine performance with concurrent guest OSes later. The results presented are the median of seven trials.

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

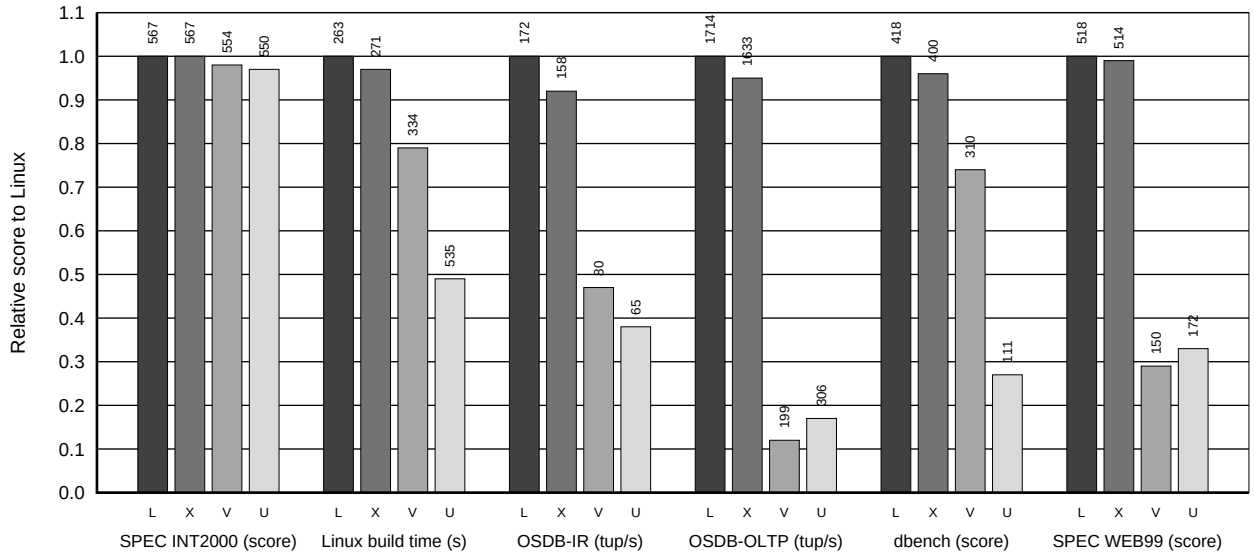

Figure 3: Relative performance of native Linux (L), XenoLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

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 Retrieval (IR) and On-Line Transaction Processing (OLTP) workloads, both measured in tuples per second. A small modification to the suite's test harness was required to produce correct results, due to a UML bug which loses virtual-timer interrupts under high load. The benchmark drives the database via PostgreSQL's native API (callable SQL) over a Unix domain socket. 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 workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations and 0.5% execute a CGI script. As the server runs it generates access and POST logs, so the disk workload is not solely read-only. Measurements therefore reflect general OS performance, including file system and network, in addition to the web server itself.

A number of client machines are used to generate load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. The benchmark is run repeatedly with different numbers of simulated users to determine the maximum number that can be supported. SPEC WEB99 defines a minimum Quality of Service that simulated users must receive in order to be 'conformant' and hence count toward the score: users must receive an aggregate bandwidth in excess of 320Kb/s over a series of requests. A warm-up phase is allowed in which the number of simultaneous clients is slowly increased, allowing servers to preload their buffer caches.

For our experimental setup we used the Apache HTTP server version 1.3.27, installing the `modspecweb99` plug-in to perform most but not all of the dynamic content generation — SPEC rules require 0.5% of requests to use full CGI, forking a separate process. Better absolute performance numbers can be achieved with the assistance of "TUX", the Linux in-kernel static content web server, but we chose not to use this as we felt it was less likely to be representative of our real-world target applications. Furthermore, although Xen's performance improves when using TUX, VMware suffers badly due to the increased proportion of time spent emulating ring 0 while executing the guest OS kernel.

SPEC WEB99 exercises the whole system. During the measurement period there is up to 180Mb/s of TCP network traffic and considerable disk read-write activity on a 2GB dataset. 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. XenoLinux 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 [29]. We used version 3.0-a3 as this addresses many of the issues regarding the fidelity of the tool raised by Seltzer's *hbench* [6]. The OS performance subset of the lmbench suite consist of 37 microbenchmarks. In the native Linux case, we present figures for both uniprocessor (L-UP) and SMP (L-SMP) kernels as we were somewhat surprised by the performance overhead incurred by the extra locking in the SMP system in many cases.

In 24 of the 37 microbenchmarks, XenoLinux performs similarly to native Linux, tracking the uniprocessor Linux kernel per-

Config	null call	null I/O	stat	open close	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
L-SMP	0.53	0.81	2.10	3.51	23.2	0.83	2.94	143	601	4k2
L-UP	0.45	0.50	1.28	1.92	5.70	0.68	2.49	110	530	4k0
Xen	0.46	0.50	1.22	1.88	5.69	0.69	1.75	**198**	**768**	**4k8**
VMW	0.73	0.83	1.88	2.99	11.1	1.02	4.63	874	2k3	10k
UML	24.7	25.1	36.1	62.8	39.9	26.0	46.0	21k	33k	58k

Table 3: `lmbench`: Processes - times in μs

Config	2p 0K	2p 16K	2p 64K	8p 16K	8p 64K	16p 16K	16p 64K
L-SMP	1.69	1.88	2.03	2.36	26.8	4.79	38.4
L-UP	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 4: `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
L-SMP	44.9	24.2	123	45.2	99.0	1.33	1.88
L-UP	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 5: `lmbench`: File & VM system latencies in μs

formance closely and outperforming the SMP kernel. In Tables 3 to 5 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 3), Xen 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 XenoLinux to batch update requests. Creating new page tables presents an ideal case: because there is no reason to commit pending updates sooner, XenoLinux can amortize each hypercall across 2048 updates (the maximum size of its batch buffer). Hence each update hypercall constructs 8MB of address space.

Table 4 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 5 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 3 is that XenoLinux 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 XenoLinux, hence underlining the dangers of taking microbenchmarks too seriously.

	TCP MTU 1500 TX	TCP MTU 1500 RX	TCP MTU 500 TX	TCP MTU 500 RX
Linux	897	897	602	544
Xen	897 (-0%)	897 (-0%)	516 (-14%)	467 (-14%)
VMW	291 (-68%)	615 (-31%)	101 (-83%)	137 (-75%)
UML	165 (-82%)	203 (-77%)	61.1(-90%)	91.4(-83%)

Table 6: `ttcp`: Bandwidth in Mb/s

4.2.1 Network performance

In order to evaluate the overhead of virtualizing the network, we examine TCP performance over a Gigabit Ethernet LAN. In all experiments we use a similarly-configured SMP box running native Linux as one of the endpoints. This enables us to measure receive and transmit performance independently. The *ttcp* benchmark was used to perform these measurements. Both sender and receiver applications were configured with a socket buffer size of 128kB, as we found this gave best performance for all tested systems. The results presented are a median of 9 experiments transferring 400MB.

Table 6 presents two sets of results, one using the default Ethernet MTU of 1500 bytes, the other using a 500-byte MTU (chosen as it is commonly used by dial-up PPP clients). The results demonstrate that the page-flipping technique employed by the XenoLinux virtual network driver avoids the overhead of data copying and hence achieves a very low per-byte overhead. With an MTU of 500 bytes, the per-packet overheads dominate. The extra complexity of transmit firewalling and receive demultiplexing adversely impact the throughput, but only by 14%.

VMware emulate a 'pcnet32' network card for communicating with the guest OS which provides a relatively clean DMA-based interface. ESX Server also supports a special 'vmxnet' driver for compatible guest OSes, which provides significant networking performance improvements.

4.3 Concurrent Virtual Machines

In this section, we compare the performance of running multiple applications in their own guest OS against running them on the same native operating system. Our focus is on the results using Xen, but we comment on the performance of the other VMMs where applicable.

Figure 4 shows the results of running 1, 2, 4, 8 and 16 copies of the SPEC WEB99 benchmark in parallel on a two CPU machine. The native Linux was configured for SMP; on it we ran multiple copies of Apache as concurrent processes. In Xen's case, each instance of SPEC WEB99 was run in its own uniprocessor Linux guest OS (along with an `sshd` and other management processes). Different TCP port numbers were used for each web server to enable the copies to be run in parallel. Note that the size of the SPEC data set required for c simultaneous connections is $(25 + (c \times 0.66)) \times 4.88$ MBytes or approximately 3.3GB for 1000 connections. This is sufficiently large to thoroughly exercise the disk and buffer cache subsystems.

Achieving good SPEC WEB99 scores requires both high throughput and bounded latency: for example, if a client request gets stalled due to a badly delayed disk read, then the connection will be classed as non conforming and won't contribute to the score. Hence, it is important that the VMM schedules domains in a timely fashion. By default, Xen uses a 5ms time slice.

In the case of a single Apache instance, the addition of a second CPU enables native Linux to improve on the score reported in section 4.1 by 28%, to 662 conformant clients. However, the

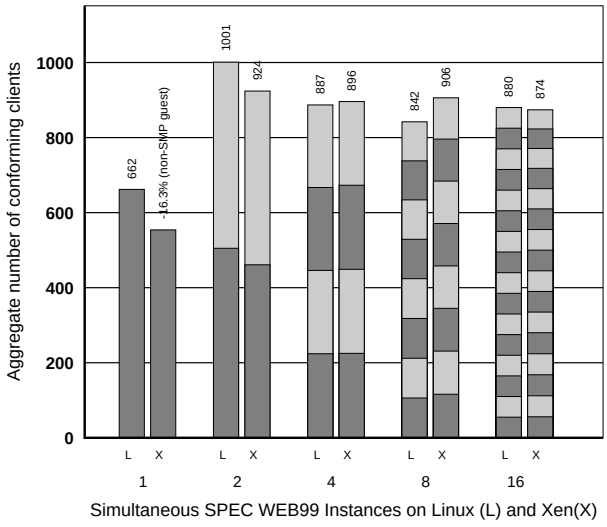

Figure 4: SPEC WEB99 for 1, 2, 4, 8 and 16 concurrent Apache servers: higher values are better.

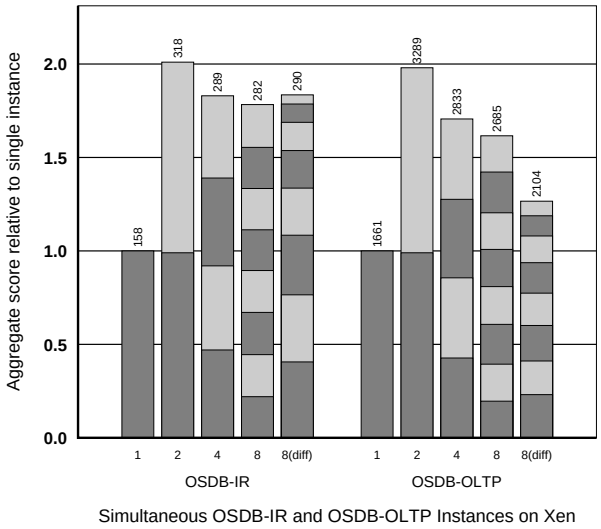

Figure 5: Performance of multiple instances of PostgreSQL running OSDB in separate Xen domains. 8(diff) bars show performance variation with different scheduler weights.

best aggregate throughput is achieved when running two Apache instances, suggesting that Apache 1.3.27 may have some SMP scalability issues.

When running a single domain, Xen is hampered by a lack of support for SMP guest OSes. However, Xen's interrupt load balancer identifies the idle CPU and diverts all interrupt processing to it, improving on the single CPU score by 9%. As the number of domains increases, Xen's performance improves to within a few percent of the native case.

Next we performed a series of experiments running multiple instances of PostgreSQL exercised by the OSDB suite. Running multiple PostgreSQL instances on a single Linux OS proved difficult, as it is typical to run a single PostgreSQL instance supporting multiple databases. However, this would prevent different users having separate database configurations. We resorted to a combination of `chroot` and software patches to avoid SysV IPC name-space clashes between different PostgreSQL instances. In contrast, Xen allows each instance to be started in its own domain allowing easy configuration.

In Figure 5 we show the aggregate throughput Xen achieves when running 1, 2, 4 and 8 instances of OSDB-IR and OSDB-OLTP. When a second domain is added, full utilization of the second CPU almost doubles the total throughput. Increasing the number of domains further causes some reduction in aggregate throughput which can be attributed to increased context switching and disk head movement. Aggregate scores running multiple PostgreSQL instances on a single Linux OS are 25-35% *lower* than the equivalent scores using Xen. The cause is not fully understood, but it appears that PostgreSQL has SMP scalability issues combined with poor utilization of Linux's block cache.

Figure 5 also demonstrates performance differentiation between 8 domains. Xen's schedulers were configured to give each domain an integer weight between 1 and 8. The resulting throughput scores for each domain are reflected in the different banding on the bar. In the IR benchmark, the weighting has a precise influence over throughput and each segment is within 4% of its expected size. However, in the OLTP case, domains given a larger share of resources to not achieve proportionally higher scores: The high level of synchronous disk activity highlights a weakness in our current disk scheduling algorithm causing them to under-perform.

4.4 Performance Isolation

In order to demonstrate the performance isolation provided by Xen, we hoped to perform a "bakeoff" between Xen and other OS-based implementations of performance isolation techniques such as resource containers. However, at the current time there appear to be no implementations based on Linux 2.4 available for download. QLinux 2.4 has yet to be released and is targeted at providing QoS for multimedia applications rather than providing full defensive isolation in a server environment. Ensim's Linux-based Private Virtual Server product appears to be the most complete implementation, reportedly encompassing control of CPU, disk, network and physical memory resources [14]. We are in discussions with Ensim and hope to be able to report results of a comparative evaluation at a later date.

In the absence of a side-by-side comparison, we present results showing that Xen's performance isolation works as expected, even in the presence of a malicious workload. We ran 4 domains configured with equal resource allocations, with two domains running previously-measured workloads (PostgreSQL/OSDB-IR and SPEC WEB99), and two domains each running a pair of extremely antisocial processes. The third domain concurrently ran a disk bandwidth hog (sustained `dd`) together with a file system intensive workload targeting huge numbers of small file creations within large directories. The fourth domain ran a 'fork bomb' at the same time as a virtual memory intensive application which attempted to allocate and touch 3GB of virtual memory and, on failure, freed every page and then restarted.

We found that both the OSDB-IR and SPEC WEB99 results were only marginally affected by the behaviour of the two domains running disruptive processes — respectively achieving 4% and 2% below the results reported earlier. We attribute this to the overhead of extra context switches and cache effects. We regard this as somewhat fortuitous in light of our current relatively simple disk scheduling algorithm, but under this scenario it appeared to pro-

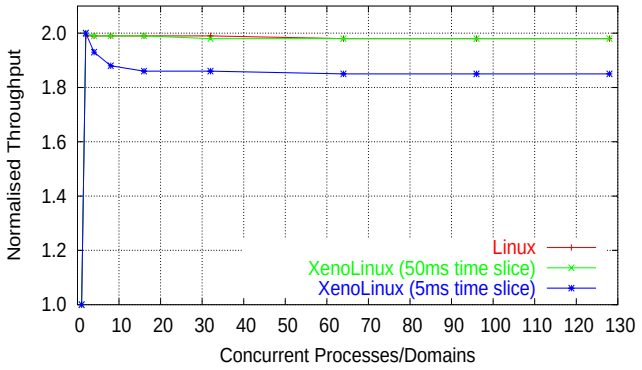

Figure 6: Normalized aggregate performance of a subset of SPEC CINT2000 running concurrently on 1-128 domains

vide sufficient isolation from the page-swapping and disk-intensive activities of the other domains for the benchmarks to make good progress. VMware Workstation achieves similar levels of isolation, but at reduced levels of absolute performance.

We repeated the same experiment under native Linux. Unsurprisingly, the disruptive processes rendered the machine completely unusable for the two benchmark processes, causing almost all the CPU time to be spent in the OS.

4.5 Scalability

In this section, we examine Xen's ability to scale to its target of 100 domains. We discuss the memory requirements of running many instances of a guest OS and associated applications, and measure the CPU performance overhead of their execution.

We evaluated the minimum physical memory requirements of a domain booted with XenoLinux and running the default set of RH7.2 daemons, along with an `sshd` and Apache web server. The domain was given a reservation of 64MB on boot, limiting the maximum size to which it could grow. The guest OS was instructed to minimize its memory footprint by returning all pages possible to Xen. Without any swap space configured, the domain was able to reduce its memory footprint to 6.2MB; allowing the use of a swap device reduced this further to 4.2MB. A quiescent domain is able to stay in this reduced state until an incoming HTTP request or periodic service causes more memory to be required. In this event, the guest OS will request pages back from Xen, growing its footprint as required up to its configured ceiling.

This demonstrates that memory usage overhead is unlikely to be a problem for running 100 domains on a modern server class machine — far more memory will typically be committed to application data and buffer cache usage than to OS or application text pages. Xen itself maintains only a fixed 20kB of state per domain, unlike other VMMs that must maintain shadow page tables etc.

Finally, we examine the overhead of context switching between large numbers of domains rather than simply between processes. Figure 6 shows the normalized aggregate throughput obtained when running a small subset of the SPEC CINT2000 suite concurrently on between 1 and 128 domains or processes on our dual CPU server. The line representing native Linux is almost flat, indicating that for this benchmark there is no loss of aggregate performance when scheduling between so many processes; Linux identifies them all as compute bound, and schedules them with long time slices of 50ms or more. In contrast, the lower line indicates Xen's throughput when configured with its default 5ms maximum scheduling 'slice'. Although operating 128 simultaneously compute bound processes on a single server is unlikely to be commonplace in our target application area, Xen copes relatively well: running 128 domains we lose just 7.5% of total throughput relative to Linux.

Under this extreme load, we measured user-to-user UDP latency to one of the domains running the SPEC CINT2000 subset. We measured mean response times of 147ms (standard deviation 97ms). Repeating the experiment against a 129^{th} domain that was otherwise idle, we recorded a mean response time of 5.4ms (s.d. 16ms). These figures are very encouraging — despite the substantial background load, interactive domains remain responsive.

To determine the cause of the 7.5% performance reduction, we set Xen's scheduling 'slice' to 50ms (the default value used by ESX Server). The result was a throughput curve that tracked native Linux's closely, almost eliminating the performance gap. However, as might be expected, interactive performance at high load is adversely impacted by these settings.

5. RELATED WORK

Virtualization has been applied to operating systems both commercially and in research for nearly thirty years. IBM VM/370 [19, 38] first made use of virtualization to allow binary support for legacy code. VMware [10] and Connectix [8] both virtualize commodity PC hardware, allowing multiple operating systems to run on a single host. All of these examples implement a full virtualization of (at least a subset of) the underlying hardware, rather than paravirtualizing and presenting a modified interface to the guest OS. As shown in our evaluation, the decision to present a full virtualization, although able to more easily support off-the-shelf operating systems, has detremental consequences for performance.

The virtual machine monitor approach has also been used by Disco to allow commodity operating systems to run efficiently on ccNUMA machines [7, 18]. A small number of changes had to be made to the hosted operating systems to enable virtualized execution on the MIPS architecture. In addition, certain other changes were made for performance reasons.

At present, we are aware of two other systems which take the paravirtualization approach: IBM presently supports a paravirtualized version of Linux for their zSeries mainframes, allowing large numbers of Linux instances to run simultaneously. Denali [44], discussed previously, is a contemporary isolation kernel which attempts to provide a system capable of hosting vast numbers of virtualized OS instances.

In addition to Denali, we are aware of two other efforts to use low-level virtualization to build an infrastructure for distributed systems. The vMatrix [1] project is based on VMware and aims to build a platform for moving code between different machines. As vMatrix is developed above VMware, they are more concerned with higher-level issues of distribution that those of virtualization itself. In addition, IBM provides a "Managed Hosting" service, in which virtual Linux instances may be rented on IBM mainframes.

The PlanetLab [33] project has constructed a distributed infrastructure which is intended to serve as a testbed for the research and development of geographically distributed network services. The platform is targeted at researchers and attempts to divide individual physical hosts into *slivers*, providing simultaneous low-level access to users. The current deployment uses VServers [17] and SILK [4] to manage sharing within the operating system.

We share some motivation with the operating system extensibility and active networks communities. However, when running over Xen there is no need to check for "safe" code, or for guaranteed termination — the only person hurt in either case is the client in question. Consequently, Xen provides a more general solution: there is no need for hosted code to be digitally signed by a trusted

compiler (as in SPIN [5]), to be accompanied by a safety proof (as with PCC [31]), to be written in a particular language (as in SafetyNet [22] or any Java-based system), or to rely on a particular middleware (as with mobile-agent systems). These other techniques can, of course, continue to be used within guest OSes running over Xen. This may be particularly useful for workloads with more transient tasks which would not provide an opportunity to amortize the cost of starting a new domain.

A similar argument can be made with regard to language-level virtual machine approaches: while a resource-managed JVM [9] should certainly be able to host untrusted applications, these applications must necessarily be compiled to Java bytecode and follow that particular system's security model. Meanwhile, Xen can readily support language-level virtual machines as applications running over a guest OS.

6. DISCUSSION AND 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 performance and resource management. We have also described and evaluated XenoLinux, a fully-featured port of a Linux 2.4 kernel that runs over Xen.

6.1 Future Work

We believe that Xen and XenoLinux are sufficiently complete to be useful to a wider audience, and so intend to make a public release of our software in the very near future. A beta version is already under evaluation by selected parties; once this phase is complete, a general 1.0 release will be announced on our project page[3].

After the initial release we plan a number of extensions and improvements to Xen. To increase the efficiency of virtual block devices, we intend to implement a shared *universal buffer cache* indexed on block contents. This will add controlled data sharing to our design without sacrificing isolation. Adding copy-on-write semantics to virtual block devices will allow them to be safely shared among domains, while still allowing divergent file systems.

To provide better physical memory performance, we plan to implement a *last-chance page cache* (LPC) — effectively a system-wide list of free pages, of non-zero length only when machine memory is undersubscribed. The LPC is used when the guest OS virtual memory system chooses to evict a clean page; rather than discarding this completely, it may be added to the tail of the free list. A fault occurring for that page before it has been reallocated by Xen can therefore satisfied without a disk access.

An important role for Xen is as the basis of the *XenoServer* project which looks beyond individual machines and is building the control systems necessary to support an Internet-scale computing infrastructure. Key to our design is the idea that resource usage be accounted precisely and paid for by the sponsor of that job — if payments are made in real cash, we can use a congestion pricing strategy [28] to handle excess demand, and use excess revenues to pay for additional machines. This necessitates accurate and timely I/O scheduling with greater resilience to hostile workloads. We also plan to incorporate accounting into our block storage architecture by creating leases for virtual block devices.

In order to provide better support for the management and administration of XenoServers, we are incorporating more thorough support for auditing and forensic logging. We are also developing additional VFR rules which we hope will allow us to detect and prevent a wide range of antisocial network behaviour. Finally, we are continuing our work on XenoXP, focusing in the first instance on writing network and block device drivers, with the aim of fully supporting enterprise servers such as IIS.

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

6.2 Conclusion

Xen provides an excellent platform for deploying a wide variety of network-centric services, such as local mirroring of dynamic web content, media stream transcoding and distribution, multiplayer game and virtual reality servers, and 'smart proxies' [2] to provide a less ephemeral network presence for transiently-connected devices.

Xen directly addresses the single largest barrier to the deployment of such services: the present inability to host transient servers for short periods of time and with low instantiation costs. By allowing 100 operating systems to run on a single server, we reduce the associated costs by two orders of magnitude. Furthermore, by turning the setup and configuration of each OS into a software concern, we facilitate much smaller-granularity timescales of hosting.

As our experimental results show in Section 4, the performance of XenoLinux over Xen is practically equivalent to the performance of the baseline Linux system. This fact, which comes from the careful design of the interface between the two components, means that there is no appreciable cost in having the resource management facilities available. Our ongoing work to port the BSD and Windows XP kernels to operate over Xen is confirming the generality of the interface that Xen exposes.

Acknowledgments

This work is supported by ESPRC Grant GR/S01894/01 and by Microsoft. We would like to thank Evangelos Kotsovinos, Anil Madhavapeddy, Russ Ross and James Scott for their contributions.

7. REFERENCES

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

[2] A. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 136–143, June 1995.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 45–58, Feb. 1999.

[4] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. SILK: Scout paths in the Linux kernel. Technical Report 2002-009, Uppsala University, Department of Information Technology, Feb. 2002.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, volume 29(5) of *ACM Operating Systems Review*, pages 267–284, Dec. 1995.

[6] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 143–156, Oct. 1997.

[8] Connectix. Product Overview: Connectix Virtual Server, 2003. `http://www.connectix.com/products/vs.html`.

[9] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 36(11):125–138, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001).

[10] 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.

[11] 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.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 211–224, Boston, MA, USA, Dec. 2002.

[13] 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.

[14] Ensim. Ensim Virtual Private Servers, 2003. `http://www.ensim.com/products/materials/datasheet_vps_051003.pdf`.

[15] K. A. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt. The Xenoserver computing infrastructure. Technical Report UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, Jan. 2003.

[16] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems, Kauai, Hawaii*, May 2003.

[17] J. Gelinas. Virtual Private Servers and Security Contexts, 2003. `http://www.solucorp.qc.ca/miscprj/s_context.hc`.

[18] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 154–169, Dec. 1999.

[19] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, Nov. 1983.

[20] S. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 73–86, Oct. 1999.

[21] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform, April 2003.

[22] A. Jeffrey and I. Wakeman. A Survey of Semantic Techniques for Active Networks, Nov. 1997. `http://www.cogs.susx.ac.uk/projects/safetynet/`.

[23] 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.

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

[25] 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.

[26] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY*, Jun 2002.

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

[28] J. MacKie-Mason and H. Varian. Pricing congestible network resources. *IEEE Journal on Selected Areas In Communications*, 13(7):1141–1149, Sept. 1995.

[29] 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.

[30] 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.

[31] G. C. Necula. Proof-carrying code. In *Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.

[32] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real Time Technology and Applications Symposium*, pages 111–120, June 1999.

[33] 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.

[34] 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.

[35] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accounted execution of untrusted code. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, 1999.

[36] 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.

[37] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 377–390, Boston, MA, USA, Dec. 2002.

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

[39] 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.

[40] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H.M.Vin. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the 8th ACM Conference on Multimedia*, Nov. 2000.

[41] D. Tennenhouse. Layered Multiplexing Considered Harmful. In Rudin and Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148. North Holland, 1989.

[42] 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.

[43] 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.

[44] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 195–210, Boston, MA, USA, Dec. 2002.