Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Xen 3.0 and the Art of Virtualization

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

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

Jun Nakajima, Asit Mallick
Intel Open Source Technology Center
{first.last}@intel.com

Abstract

The Xen Virtual Machine Monitor will soon be undergoing its third major release, and is maturing into a stable, secure, and full-featured virtualization solution for Linux and other operating systems. Xen has attracted considerable development interest over the past year, and consequently the 3.0 release includes many exciting new features. This paper provides an overview of the major new features, including VM relocation, device driver isolation, support for unmodified operating systems, and new hardware support for both x86/64 and IA-64 processors.

1 VM Relocation

While many server applications may be very long-lived, the hardware that it runs on will invariably need service from time to time. A major benefit of virtualization is the ability to relocate a *running* operating system instance from one physical host to another. Relocation allows a physical host to be unloaded so that hardware may be serviced, it allows coarse-grained load-balancing in a cluster environment, and it allows servers to move closer to the users that they serve. By *pre-copying* VM state to the destination host while it is still running, relocation down-time can be made very small—experiments relocating a running Quake server have achieved repeatable relocation times with outages of less than 100ms.

In the following subsections we describe some of the implementation details of our pre-copying approach. We describe how we use dynamic network rate-limiting to effectively balance network contention against OS downtime. We then proceed to describe how we ameliorate the effects of rapid page dirtying, and show results for the relocation of a running Quake 3 server.

1.1 Managing Relocation

Relocation is performed by daemons running in the management VMs of the source and destination hosts. These are responsible for creating a new VM on the destination machine, and co-ordinating transfer of live system state over the network.

When transferring the memory image of the still-running OS, the control software performs

rounds of copying in which it performs a complete scan of the VM's memory pages. Although in the first round all pages are transferred to the destination machine, in subsequent rounds this copying is restricted to pages that were dirtied during the previous round, as indicated by a *dirty bitmap* that is copied from Xen at the start of each round.

During normal operation the page tables managed by each guest OS are the ones that are walked by the processor's MMU to fill the TLB. This is possible because guest OSes are exposed to real physical addresses and so the page tables they create do not need to be mapped to physical addresses by Xen.

To log pages that are dirtied, Xen inserts *shadow page tables* underneath the running OS. The shadow tables are populated on demand by translating sections of the guest page tables. Translation is very simple for dirty logging: all page-table entries (PTEs) are initially read-only mappings in the shadow tables, regardless of what is permitted by the guest tables. If the guest tries to modify a page of memory, the resulting page fault is trapped by Xen. If write access is permitted by the relevant guest PTE then this permission is extended to the shadow PTE. At the same time, we set the appropriate bit in the VM's dirty bitmap.

When the bitmap is copied to the control software at the start of each pre-copying round, Xen's bitmap is cleared and the shadow page tables are destroyed and recreated as the relocatee OS continues to run. This causes all write permissions to be lost: all pages that are subsequently updated are then added to the now-clear dirty bitmap.

When it is determined that the pre-copy phase is no longer beneficial, the OS is sent a control message requesting that it suspend itself in a state suitable for relocation. This causes the OS to prepare for resumption on the destination machine; Xen informs the control software once the OS has done this. The dirty bitmap is scanned one last time for remaining inconsistent memory pages, and these are transferred to the destination together with the VM's checkpointed CPU-register state.

Once this final information is received at the destination, the VM state on the source machine can safely be discarded. Control software on the destination machine scans the memory map and rewrites the guest's page tables to reflect the addresses of the memory pages that it has been allocated. Execution is then resumed by starting the new VM at the point that the old VM checkpointed itself. The OS then restarts its virtual device drivers and updates its notion of wallclock time.

1.2 Dynamic Rate-Limiting

It is not always appropriate to select a single network bandwidth limit for relocation traffic. Although a low limit avoids impacting the performance of running services, analysis showed that we must eventually pay in the form of an extended downtime because the hottest pages in the writable working set are not amenable to pre-copy relocation. The downtime can be reduced by increasing the bandwidth limit, albeit at the cost of additional network contention.

Our solution to this impasse is to dynamically adapt the bandwidth limit during each pre-copying round. The administrator selects a minimum and a maximum bandwidth limit. The first pre-copy round transfers pages at the minimum bandwidth. Each subsequent round counts the number of pages dirtied in the previous round, and divides this by the duration of the previous round to calculate the *dirtying rate*. The bandwidth limit for the next round is then determined by adding a constant increment to the previous round's dirtying rate—we

Iterative Progress of Live Migration: Quake 3 Server
6 Clients, 64MB VM
Total Data Transmitted: 88MB (x1.37)

Area of Bars:
- VM memory transferred
- Memory dirtied during this iteration

The final iteration in this case leaves only 148KB of data to transmit. In addition to the 20ms required to copy this last round, an additional 40ms are spent on start-up overhead. The total downtime experienced is 60ms.
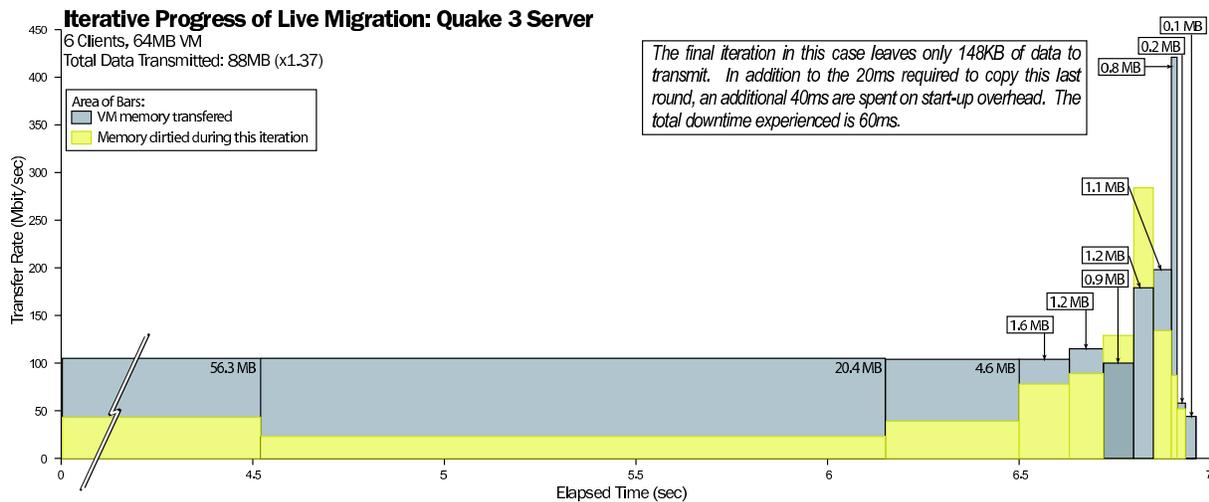
Figure 1: Results of relocating a running Quake 3 server VM.

have empirically determined that 50Mbit/sec is a suitable value. We terminate pre-copying when the calculated rate is greater than the administrator's chosen maximum, or when less than 256KB remains to be transferred. During the final stop-and-copy phase we minimize service downtime by transferring memory at the maximum allowable rate.

Using this adaptive scheme results in the bandwidth usage remaining low during the transfer of the majority of the pages, increasing only at the end of the relocation to transfer the hottest pages in the WWS. This effectively balances short downtime with low average network contention and CPU usage.

1.3 Rapid Page Dirtying

Analysis shows that every OS workload has some set of pages that are updated extremely frequently, and which are therefore not good candidates for pre-copy relocation even when using all available network bandwidth. We observed that rapidly-modified pages are very likely to be dirtied again by the time we attempt to transfer them in any particular pre-copying round. We therefore periodically 'peek' at the current round's dirty bitmap and transfer only those pages dirtied in the previous round that have not been dirtied again at the time we scan them.

We further observed that page dirtying is often physically *clustered*—if a page is dirtied then it is disproportionally likely that a close neighbour will be dirtied soon after. This increases the likelihood that, if our peeking does not detect one page in a cluster, it will detect none. To avoid this unfortunate behaviour we scan the VM's physical memory space in a pseudo-random order.

1.4 Low-Latency Server: Quake 3

A representative application for hosting environments is a multiplayer on-line game server. To determine the effectiveness of our approach in this case we configured a virtual machine with 64MB of memory running a Quake 3 server. Six players joined the game and started to play within a shared arena, at which point we initiated a relocation to another machine. A detailed analysis of this relocation is shown in Figure 1.

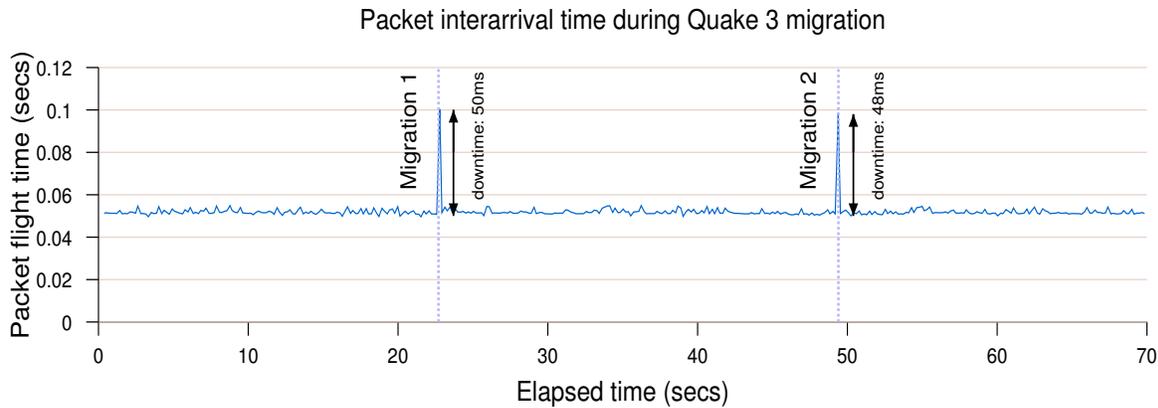Packet interarrival time during Quake 3 migration

Figure 2: Effect on packet response time of relocating a running Quake 3 server VM.

We were able to perform the live relocation with a total downtime of 60*ms*. To determine the effect of relocation on the live players, we performed an additional experiment in which we relocated the running Quake 3 server twice and measured the inter-arrival time of packets received by clients. The results are shown in Figure 2. As can be seen, from the client point of view relocation manifests itself as a transient increase in response time of 50*ms*. In neither case was this perceptible to the players.

2 Device Virtualization

Xen's strong isolation guarantees have proved very useful in solving two major problems with device drivers: driver availability and reliability. Xen is capable of allowing individual virtual machines to have direct access to specific pieces of hardware. We have taken the approach of using a single virtual machine to run the physical driver for a device (such as a disk or network interface) and then export a virtualized version of the device to all of the other guest OSes that are running on the host. This approach means that a device need only be supported on a single platform (Linux, for instance), and may be available to all the OSes

that Xen runs. Each guest implements an idealized disk and network device, which are capable of connecting to the hardware specific driver in an isolated *device domain*. This approach has the added benefit of making drivers, which are a major source of bugs in operating systems, more reliable. By running a driver in its own VM, driver crashes are limited to the driver itself—other applications may continue to run. Device domains can even be rebooted to recover failed drivers, and result in downtimes on the order of hundreds of miliseconds in cases where the entire machine would previously have crashed completely.

This approach will no doubt sound familiar to anyone who has worked with microkernels in the past—Xen's isolation achieves a similar fragmentation of OS subsystems. One major difference between Xen and historical work on microkernels is that we have forgone the architecturally pure fixation on IPC mechanisms in favour of a generalized, shared-memory ring-based communication primitive that is able to achieve very high throughputs by batching requests.

To achieve driver isolation, we restrict access privileges to device I/O registers (whether memory-mapped or accessed via explicit I/O

ports) and interrupt lines. Furthermore, where it is possible within the constraints of existing hardware, we protect against device misbehavior by isolating device-to-host interactions. Finally, we virtualize the PC's hardware *configuration space*, restricting each driver's view of the system so that it cannot see resources that it cannot access.

2.1 I/O Registers

Xen ensures memory isolation amongst domains by checking the validity of address-space updates. Access to a memory-mapped hardware device is permitted by extending these checks to allow access to non-RAM page frames that contain memory-mapped registers belonging to the device. Page-level protection is sufficient to provide isolation because register blocks belonging to different devices are conventionally aligned on no less than a page boundary.

In addition to memory-mapped I/O, many processor families provide an explicit I/O-access primitive. For example, the x86 architecture provides a 16-bit I/O port space to which access may be restricted on a per-port basis, as specified by an access bitmap that is interpreted by the processor on each port-access attempt. Xen uses this hardware protection by rewriting the port-access bitmap when context-switching between domains.

2.2 Interrupts

Whenever a device's interrupt line is asserted it triggers execution of a stub routine within Xen rather than causing immediate entry into the domain that is managing that device. In this way Xen retains tight control of the system by *scheduling* execution of the domain's interrupt service routine (ISR). Taking the interrupt in Xen also allows a timely acknowledgement response to the interrupt controller (which is always managed by Xen) and allows the necessary address-space switch if a different domain is currently executing. When the correct domain is scheduled it is delivered an asynchronous *event notification* which causes execution of the appropriate ISR.

Xen notifies each domain of asynchronous events, including hardware interrupts, via a general-purpose mechanism called *event channels*. Each domain can be allocated up to 1024 event channels, each of which comprises a pair of bit flags in a memory page shared between the domain and Xen. The first flag is used by Xen to signal that an event is *pending*. When an event becomes pending Xen schedules an asynchronous upcall into the domain; if the domain is blocked then it is moved to the run queue. Unnecessary upcalls are avoided by triggering a notification only when an event first becomes pending: further settings of the flag are then ignored until after it is cleared by the domain.

The second event-channel flag is used by the domain to *mask* the event. No notification is triggered when a masked event becomes pending: no asynchronous upcall occurs and a blocked domain is not woken. By setting the mask before clearing the pending flag, a domain can prevent unnecessary upcalls for partially-handled event sources.

To avoid unbounded reentrancy, a level-triggered interrupt line must be masked at the interrupt controller until all relevant devices have been serviced. After handling an event relating to a level-triggered interrupt, the domain must call *down* into Xen to unmask the interrupt line. However, if an interrupt line is not shared by multiple devices then Xen can usually safely reconfigure it as edge-triggering, obviating the need for unmask downcalls.

When an interrupt line is shared by multiple hardware devices, Xen must delay unmasking the interrupt until a downcall is received from every domain that is managing one of the devices. Xen cannot guarantee perfect isolation of a domain that is allocated a shared interrupt: if the domain never unmasks the interrupt then other domains can be prevented from receiving device notifications. However, shared interrupts are rare in server-class systems which typically contain IRQ-steering and interrupt-controller components with enough pins for every device. The problem of sharing is set to disappear completely with the introduction of message-based interrupts as part of PCI Express [1].

2.3 Device-to-Host Interactions

As well as preventing a device driver from circumventing its isolated environment, we must also protect against possible misbehavior of the hardware itself, whether due to inherent design flaws or misconfiguration by the driver software. The two general types of device-to-host interaction that we must consider are assertion of interrupt lines, and accesses to host memory space.

Protecting against arbitrary interrupt assertion is not a significant issue because, except for shared interrupt lines, each hardware device has its own separately-wired connection to the interrupt controller. Thus it is physically impossible for a device to assert any interrupt line other than the one that is assigned to it. Furthermore, Xen retains full control over configuration of the interrupt controller and so can guard against problems such as 'IRQ storms' that could be caused by repeated cycling of a device's interrupt line.

The main 'protection gap' for devices, then, is that they may attempt to access arbitrary ranges of host memory. For example, although a device driver is prevented from using the CPU to write to a particular page of system memory (perhaps because the page does not belong to the driver), it may instead program its hardware device to perform a DMA to the page. Unfortunately there is no good method for protecting against this problem with current hardware as it is infeasible for Xen to validate the programming of DMA-related device registers. Not only would this require intimate knowledge of every device's DMA engine, it also would not protect against bugs in the hardware itself: buggy hardware would still be able to access arbitrary system memory.

A full implementation of this aspect of our design requires integration of an IOMMU into the PC chipset—a feature that is expected to be included in commodity chipsets in the very near future. Similar to the processor's MMU, this translates the addresses requested by a device into valid host addresses. Inappropriate host addresses are not accessible to the device because no mapping is configured in the IOMMU. In our design, Xen would be responsible for configuring the IOMMU in response to requests from domains. The required validation checks are identical to those required for the processor's MMU; for example, to ensure that the requesting domain owns the page frame, and that it is safe to permit arbitrary modification of its contents.

2.4 Hardware Configuration

The PCI standard defines a generic *configuration space* through which PC hardware devices are detected and configured. Xen restricts each domain's access to this space so that it can read and write registers belonging only to a device that it owns. This serves a dual purpose: not only does it prevent cross-configuration of other domains' devices, but it also restricts the

domain's view so that a hardware probe detects only devices that it is permitted to access.

The method of access to the configuration space is system-dependent, and the most common methods are potentially unsafe (either protected-mode BIOS calls, or a small I/O-port 'window' that is shared amongst all device spaces). Domains are therefore not permitted direct access to the configuration space, but are forced to use a virtualized interface provided by Xen. This has the advantage that Xen can perform arbitrary validation and translation of access requests. For example, Xen disallows any attempt to change the base address of an I/O-register block, as the new location may conflict with other devices.

2.5 Device Channels

Guest OSs access devices via *device channel* links with isolated driver domains (IDDs). The channel is a point-to-point communication link through which each party can asynchronously send messages to the other. Channels are established by using a privileged *device manager* to introduce an IDD to a guest OS, and vice versa. To facilitate this, the device manager automatically establishes an initial control channel with each domain that it creates. Figure 3 shows a guest OS requesting a data transfer through a device channel. The individual steps involved are discussed later in this section.

Xen itself has no concrete notion of a control or device channel. Messages are communicated via shared memory pages that are allocated by the guest OS but are simultaneously mapped into the address space of the IDD or device manager. For this purpose, Xen permits restricted *sharing* of memory pages between domains.

The sharing mechanism provided by Xen differs from traditional application-level shared
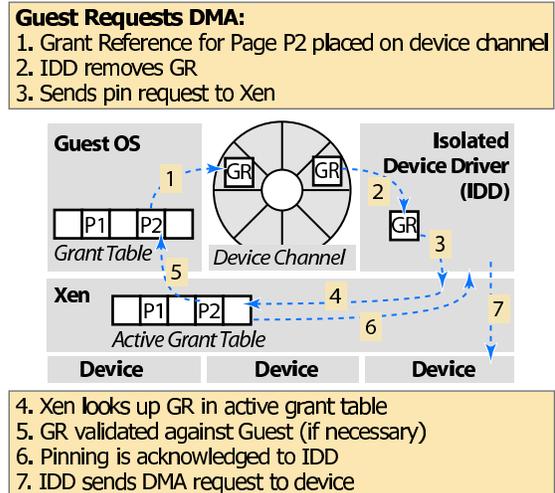

Figure 3: Using device channel to request a data transfer.

memory in two key respects: shared mappings are *asymmetric* and *transitory*. Each page of memory is owned by at most one domain at any time and, with the assistance of Xen and the device manager, that owner may force reclamation of mappings from within other misbehaving domains.

To add a foreign mapping to its address space, a domain must present a valid *grant reference* to Xen in lieu of the page number. A grant reference comprises the identity of the domain that is granting mapping permission, and an index into that domain's private *grant table*. This table contains tuples of the form $(grant, D, P, R, U)$ which permit domain D to map page P into its address space; asserting the boolean flag R restricts D to read-only mappings. The flag U is written by Xen to indicate whether D currently maps P (i.e., whether the grant tuple is *in use*).

When Xen is presented with a grant reference (A, G) by a domain B, it first searches for index G in domain A's *active grant table* (AGT), a table only accessible by Xen. If no match is found, Xen reads the appropriate tuple from domain A's grant table and checks that T=*grant*

and *D=B*, and that *R=false* if *B* is requesting a writable mapping. Only if the validation checks are successful will Xen copy the tuple into the AGT and mark the grant tuple as in use.

Xen tracks grant references by associating a usage count with each AGT entry. When a foreign mapping is created with reference to an existing AGT entry, Xen increments its count. The grant reference cannot be reallocated or reused by the granting domain until the foreign domain destroys all mappings that were created with reference to it.

Although it is clear that this mechanism allows strict checking of foreign mappings when they are created, it is less obvious how these mappings might be revoked. For example, if a faulty IDD stops responding to service requests then guest OSs could end up owning unusable memory pages. We handle the possibility of driver failure by taking a deadline-based approach: if a guest observes that a grant table entry is still marked as in use when it determines that it ought to have been relinquished (e.g., because it requested that the device channel should be destroyed), then it signals a potential domain failure to the device manager.

The device manager checks whether the specified grant reference exists in the notifying domain's AGT and, if so, sets a deadline by which the suspect domain must relinquish the stale mappings. If a registered deadline passes but stale mappings still exist then Xen notifies the device manager. At this point the device manager may choose to destroy and restart the driver, thereby forcibly reclaiming the foreign mappings.

2.6 Descriptor Rings

I/O descriptor rings are used for asynchronous transfers between a guest OS and an IDD. Ring updates are based around two pairs of producer-consumer indexes: the guest OS places service requests onto the ring, advancing a request-producer index, while the IDD removes these requests for handling, advancing an associated request-consumer index. Responses are queued onto the same ring as requests, albeit with the IDD as producer and the guest OS as consumer. A unique identifier on each request/response allows reordering if the IDD so desires.

The guest OS and IDD use a shared *inter-domain* event channel to send asynchronous notifications of queued descriptors. An inter-domain event channel is similar to the interrupt-attached channels described in Section 2.2. The main differences are that notifications are triggered by the domain attached to the opposite end of the channel (rather than Xen), and that the channel is *bidirectional*: each end may independently notify or mask the other.

We decouple the production of requests or responses on a descriptor ring from the notification of the other party. For example, in the case of requests, a guest may enqueue multiple entries before notifying the IDD; in the case of responses, a guest can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to independently balance its latency and throughput requirements.

2.7 Data Transfer

Although storing I/O data directly within ring descriptors is a suitable approach for low-bandwidth devices, it does not scale to high-performance devices with DMA capabilities. When communicating with this class of device, data buffers are instead allocated out-of-band by the guest OS and indirectly referenced within I/O descriptors.

When programming a DMA transfer directly to or from a hardware device, the IDD must first *pin* the data buffer. We enforce driver isolation by requiring the guest OS to pass a grant reference in lieu of the buffer address: the IDD specifies this grant reference when pinning the buffer. Xen applies the same validation rules to pin requests as it does for address-space mappings. These include ensuring that the memory page belongs to the correct domain, and that it isn't attempting to circumvent memory-management checks (for example, by requesting a device transfer directly into its page tables).

Returning to the example in Figure 3, the guest's data-transfer request includes a grant reference *GR* for a buffer page P_2. The request is dequeued by the IDD which sends a pin request, incorporating GR, to Xen. Xen reads the appropriate tuple from the guest's grant table, checks that P_2 belongs to the guest, and copies the tuple into the AGT. The IDD receives the address of P_2 in the pin response, and then programs the device's DMA engine.

On systems with protection support in the chipset (Section 2.3), pinning would trigger allocation of an entry in the IOMMU. This is the only modification required to enforce safe DMA. Moreover, this modification affects only Xen: the IDDs are unaware of the presence of an IOMMU (in either case pin requests return a bus address through which the device can directly access the guest buffer).

2.8   Device Sharing

Since Xen can simultaneously host many guest OSs it is essential to consider issues arising from device sharing. The control mechanisms for managing device channels naturally support multiple channels to the same IDD. We describe below how our block-device and network IDDs support multiplexing of service requests from different clients.

Within our block-device driver we service *batches* of requests from competing guests in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disc controller. This balances good throughput with reasonably fair access. We take a similar approach for network transmission, where we implement a credit-based scheduler allowing each device channel to be allocated a bandwidth share of the form *x* bytes every *y* microseconds. When choosing a packet to queue for transmission, we round-robin schedule amongst all the channels that have sufficient credit.

A shared high-performance network-receive path requires careful design because, without demultiplexing packets in hardware [2], it is not possible to DMA directly into a guest-supplied buffer. Instead of copying the packet into a guest buffer after performing demultiplexing, we instead *exchange ownership* of the page containing the packet with an unused page provided by the guest OS. This avoids copying but requires the IDD to queue page-sized buffers at the network interface. When a packet is received, the IDD immediately checks its demultiplexing rules to determine the destination channel—if the guest has no pages queued to receive the packet, it is dropped.

3   Supporting Unmodified OSes

Xen's original goal was to provide fast virtualization, which was achieved by 'paravirtualizing' guest OSes. The downside of paravirtualization is that it requires modification of the guest OS source code—an approach which is untenable for closed-source operating systems.

The alternative, full virtualization of the hardware platform, has traditionally been very difficult on the x86 processor architecture. However, new processor extensions promised by AMD and Intel provide hardware assistance which makes full virtualization much easier to provide.

Preliminary support for Intel Virtualization Technology for x86 processors (VT-x) is already checked into the Xen repository. This provides a 'virtual processor' abstraction to the guest OS which, for example, can transparently notify Xen of any attempt to execute instructions that would modify privileged processor state. While these hardware extensions make transparent virtualization easier, Xen still bears responsibility for device management and enforcing isolation of shared resources such as CPU time and memory.

3.1 VT-x architecture overview

VT-x augments the x86 architecture with two new forms of CPU operation: VMX root operation and VMX non-root operation. Xen runs in VMX root operation, while guests run in VMX non-root operation. Both forms of operation support all four privilege levels (rings 0 through 3), allowing a guest OS to appear to run at its usual 'most privileged' level. VMX root operation is similar to x86 without VT-x. Software running in VMX non-root operation is deprivileged in certain ways, regardless of privilege level.

VT-x defines two new transitions: a *VM entry* that transitions from Xen root operation to guest non-root operation, and a *VM exit* which does the opposite transition. Both VM entries and VM exits load CR3 (the base address of the page-table hierarchy) allowing Xen and the guest to run in different address spaces. VT-x also defines a virtual-machine control structure (VMCS) that manages VM entries and exits and defines processor behavior during non-root execution.

Processor behavior changes substantially in VMX non-root operation. Most importantly, many instructions and events cause VM exits. Some instructions cannot be executed in VMX non-root operation because they cause VM exits unconditionally; these include CPUID, MOV from CR3, RDMSR, and WRMSR. Other instructions, interrupts, and exceptions can be configured to cause VM exits conditionally, using VM-execution control fields in the VMCS.

VM entry loads processor state from the guest-state area of the VMCS. Xen can optionally configure VM entry to inject an interrupt or exception. The CPU effects this injection using the guest IDT, just as if the injected event had occurred immediately after VM entry. This feature removes the need for Xen to emulate delivery of these events. VM exits save processor state into the guest-state area and load processor state from the host-state area. All VM exits use a common entry point into Xen. To simplify the design of Xen, every VM exit saves into the VMCS detailed information specifying the reason for the exit; many exits also record an exit qualification, which provides further details.

3.2 VT-x Support in Xen

The three major components for adding support of VT-x and running unmodified OS in Xen are:

1. Extensions to the Xen hypervisor

2. Device models that emulate the PC platform

3. Administrator control panel

The hypervisor extensions involve adding support for the specific hardware features and instruction opcodes added by VT-x, and extensions to the user-space control tools for building and controlling fully-virtualized guests. Device models provide emulation of the PC platform devices for a VMX domain. The software models emulate all the hardware-level programming interfaces that a normal device driver uses to perform I/O operation, and submit requests to a physical device on the VMX domain's behalf.

QEMU and Bochs are two open source PC platform emulators that provided most of the functionality we needed for I/O emulation for VMX domains. Our basic design has been to run the device models in domain 0 user space and run one process for each VMX domain. We needed to remove all CPU emulation code from these emulators and modify the code that emulated physical memory (RAM). Normally, they allocate a large array to emulate the physical memory. We modified the code to map all the physical memory allocated to the VMX domain.

An example of I/O request handling from VMX guest is as follows:

1. VM exit due to an I/O access.

2. Xen decodes the instruction.

3. Xen constructs an I/O request describing the event.

4. Xen sends the request to the device-model process in domain 0.

5. When reading from a device register, the VMX domain is blocked until a response is received from the device model.

4   New Architectures

Xen was originally designed and implemented to support the x86 architecture. As interest in Xen has increased, several organisations have expressed interest in using Xen as a common hypervisor for other hardware platforms. The last year has seen fervent development of architectural support for both x86/64 and IA-64.

In addition to x86/64 and IA-64, ports of Xen are underway to the IBM Power platform and to both of the upcoming fully virtualized versions of x86, Intel's VT (described in the previous section) and AMD's Pacifica/SVM. Our experience with IA-64 supports our belief that Xen will successfully accommodate these new architectures and any others that come along in the future.

4.1   x86/64

When extending Xen to support the x86/64 architecure, we kept in mind that the platform is largely identical to x86/32, differing only in some of the details of the processor architecture. For example, processor registers are extended to 64 bits and the page-table format is extended to support the larger address space. Fortunately, the hardware platfrom is largely identical: for example, sharing the same I/O bus and chipset implementations.

This led us to implement x86/64 support as a sub-architecture of the existing x86 target. Large swathes of code are shared between sub-architectures, with the main differences being in assembly-code stubs and page-table management.

From a guest perspective, the most interesting change presented by x86/64 is the modified protection model. x86/64 provides very limited segment-level protection which makes

it impossible to protect the hypervisor, running in ring 0, from a guest kernel running in ring 1. This architectural change necessitates running both the guest kernel and applications in ring 3, and raises the problem of protecting one from the other.

The solution is to run the guest kernel in a different address space (i.e., on different page tables) from its applications. When forking a new process, the guest kernel creates two new page tables: one that is used in application context, and the other in kernel context. The kernel page table contain all the same mappings as the application page table but also include a mapping of the kernel address space. All transitions between application and guest-kernel contexts must pass via Xen, which automatically switches between the two page tables.

4.2 IA-64

As of this writing, Xen/ia64 is between its alpha release and beta release. All basic hypervisor capability is present: Domain 0 runs solidly as a 'demoted' guest OS, utilizing all devices while booting to a full graphical console and executing all Linux/ia64 applications unchanged. Multiple guest domains can be launched, but virtual I/O functionality is not finished so any unprivileged domain boots to the point where init fails to find a root disk, then panics and reboots in an infinite cycle. SMP support is not yet present, either in the hypervisor itself or in the guest.

Full functionality in Xen/ia64 is expected later this year, but the port is sufficiently complete to illustrate some similarities and differences that establish credibility that Xen will prove widely portable:

1. Hardware-walked page tables must be carefully managed in Xen/x86 and, indeed, handling page tables is one of the most complex parts of Xen, requiring a fair amount of code in the hypervisor and non-trivial changes in the paravirtualization of guests. On ia64, hardware page-table walking is still necessary for performance, but can be much more easily diverted to hypervisor-managed page tables. The difference is completely hidden from common code and implemented in the arch-specific layer.

2. Like the x86 architecture, ia64 is not fully virtualizable—certain instructions have different results when executed at different privilege levels. Both Xen architectures 'demote' the guest OS and provide an interface to handle these privilege-sensitive operations.

3. While the x86 has a small state vector, the ia64 architecture has well over 500 registers and two stacks that must be carefully managed for each thread. Linux solves this elegantly with multiple state staging areas and lazy save/restore to optimize kernel entry and exit and thread switching. Recognizing the similarity between Linux threads and Xen domains allows most of the Linux code to be directly reusable.

4. The page size on x86 is 4kB. Modern versions of x86 chips support a larger page size, but its use is limited. IA-64 supports nine different page sizes and a guest OS may use all of them simultaneously. Thus, Xen/ia64 must manage this additional complexity. Again, this is safely hidden in Xen through asm macros and arch-specific modules.

5 Conclusion

In this paper, we have presented a brief overview of the major new features in Xen 3.0 including VM relocation, device driver isolation, support for unmodified operating systems, and new hardware support for both x86/64 and IA-64 processors. Xen is quickly maturing into an enterprise-class VMM and is currently being used in production environments around the globe.

References

[1] PCI Express base specification 1.0a. PCI-SIG, 2002.

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