# Proceedings of the Linux Symposium

## Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

## Conference Organizers

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

## Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

# The Ongoing Evolution of Xen

Ian Pratt
*XenSource*
ian@xensource.com

Dan Magenheimer
*HP*
dan.magenheimer@hp.com

Hollis Blanchard
*IBM*
hollisb@us.ibm.com

Jimi Xenidis
*IBM*
jimix@watson.ibm.com

Jun Nakajima
*Intel*
jun.nakajima@intel.com

Anthony Liguori
*IBM*
aliguori@us.ibm.com

## Abstract

Xen 3 was released in December 2005, bringing new features such as support for SMP guest operating systems, PAE and x86_64, initial support for IA64, and support for CPU hardware virtualization extensions (VT/SVM). In this paper we provide a status update on Xen, reflecting on the evolution of Xen so far, and look towards the future. We will show how Xen's VT/SVM support has been unified and describe plans to optimize our support for unmodified operating systems. We discuss how a single 'xenified' kernel can run on bare metal as well as over Xen. We report on improvements made to the Itanium support and on the status of the ongoing PowerPC port. Finally we conclude with a discussion of the Xen roadmap.

## 1 Introduction

Xen is an open-source *para-virtualizing* virtual machine monitor or *hypervisor*. Xen can securely execute multiple virtual machines on a single physical system with close-to-native performance. Xen also enables advanced features such as dynamic virtual memory- and CPU-hotplug, and the live relocation of virtual machines between physical hosts. The most recent major release of Xen, Xen 3.0.0, took place on 5 December 2005.

In this paper we discuss some of the highlights of the work involved in the ongoing evolution of Xen 3. In particular we cover:

- HVM: the unified abstraction layer which allows Xen to seamlessly support both Intel and AMD processor virtualization technologies;
- the work to allow a single Linux kernel binary image to run both on Xen and on the bare metal with minimal performance and complexity costs;
- the progress made in the IA64 port of Xen, which has been much improved over the last year; and

- the ongoing port of Xen to the PowerPC architecture both with and without firmware enhancements.

Finally we look towards the future development of key technologies for Xen.

## 2 Hardware Virtual Machines

Although Xen has excellent performance, the paravirtualization techniques it applies require the modification of the guest operating system kernel code. While this is of course possible for open source systems such as Linux, it is an issue when attempting to host unmodifiable proprietary operating systems such as MS Windows.

Fortunately, recent processors from Intel and AMD sport extensions to enable the safe and efficient virtualization of unmodified operating systems. In Xen 3.0.0, initial support for Intel's VT-x extensions was introduced; later in Xen 3.0.2, further support was added to enable AMD's SVM extensions.

Although the precise details of VT-x and SVM differ, in many aspects they are quite similar. Noticing this, we decided that we could best support both technologies by introducing a layer of abstraction: the *hardware virtual machine* (HVM) interfaces. The design of the HVM architecture involved people from Intel, AMD, XenSource, and IBM, but the primary author of the code was IBM's Leendert van Doorn.

At the core of HVM is an indirection table of function pointers ( `struct hvm_function_table`). This provides mechanisms to create and destroy the additional resources required for a HVM domain (e.g. a `vmcs` on VT-x or a `vmcb` on SVM); to load and store guest state (user registers, control registers, `msrs`, etc.); and to interrogate guest operating modes (e.g. to determine if the guest is running in real mode or protected mode). By using this indirection mechansism, most of the Xen code is isolated from the details of which underlying hardware virtualization implementation is in use.

Underneath this interface, vendor-specific code invokes the appropriate HVM functions to deal with intercepts (e.g. when I/O or MMIO operations occur), and can share a considerable amount of common code (e.g. the implementation of shadow page tables, and interfacing with I/O emulation).

In the following we look first at a detailed case study—the implementation of HVM guests on 64-bit x86 platforms—and then look toward future work.

### 2.1 x86-64

One of the notable things in x86-64 Xen 3.0 is that we now support three types of HVM guests, including:

- x86-32 (2-level page tables),
- x86-32 PAE (3-level page tables), and
- x86-64 (4-level page tables).

In this section, we discuss the challenges, and we present the approaches we took for the x86-64 Xen.

The HVM architecture allows 64-bit VMMs (Virtual Machine Monitor) to run 32-bit guests securely by setting up the HVM control structure as such. Given such hardware support, we needed to work on the two major areas:

1. shadow page tables: x86-32 Xen supported only guests with 2-level page tables, and we needed to significantly extend the code to support various paging models in the single code base.

2. SMP support: SMP is the default configuration on many x86-64 and PAE systems. Supporting SMP HVM guests turns out to be far from trivial.

### 2.1.1 Overview of Shadow Page Tables in Xen 3.0

A shadow page table is the active or native page table (i.e., with entries containing *machine physical* page frame numbers) for a guest, and it is constructed by Xen to reflect the guest's operations on the guest page tables while ensuring that the guest address space is isolated securely.

A guest page table is managed and updated by the guest as it were in the native system, and it is inactive or virtual in terms of address translations (i.e., with entries containing *guest physical* page frame numbers). This is done by intercepting `mov` from/to `cr3` instructions by hardware virtualization. In other words, the value read from or written to `cr3`is virtual in HVM guests. This implies that the paging levels could even be different in the shadow and guest page tables.

From the performance point of view, shadow page table handling is critical because page faults are very frequent in memory intensive workloads. A rudimentary (and very slow) implementation is to construct shadow page tables from scratch every time the guest updates `cr3` or flushes the TLB(s). This is not efficient because the shadow page tables are lost when the guest kernel schedules the next processes to run. Frequent context switches would cause significant performance regression.

If one can efficiently tell which guest page table entries have been modified since the last TLB flush operations, we can reuse the previous shadow page tables by updating the only the page table entries that have been modified.

The key technique that we used in Xen is summarized as follows:

1. When allocating a shadow page upon #PF from the guest, write protect the corresponding guest page table page. By write-protecting the guest page tables, we can detect attempts to modify page tables.

2. Upon #PF against a guest page table page, we save a 'snapshot' of the page; give write permission to the page; and add the page is added to an 'out of sync list' along with information relating to the access attempt (e.g. which address, etc.).

3. Next we give write-permission to the page, thus allowing the guest to directly update the page table page.

4. When the guest executes an operation that results in the flush TLB operation, reflect all the entries on the "out of sync list" to the shadow page table. By comparing the snapshot and the current page in the guest page table, we can update the shadow page table efficiently by checking if the page frame numbers in the guest page tables (and thus the corresponding shadow entries) are valid.

### 2.1.2 2-Level Guest Page Table and 3-Level Shadow Page Table

The issue with running a guest with 2-level page tables is that such page tables can specify only page frames below 4GB. If we simply use a 2-level page table for the shadow page table, the page frames that we can use are restricted on any machine with more than 4GB memory.

The technique we use is to run such guests in PAE mode that utilizes 3-level (shadow) page tables, but retaining the the illusion that they are handling 2-level page tables. This complicates the shadow page table handling code in a number of ways:

- The size of a PTE (Page Table Entry) is different: 4-bytes in the guest (2-level) page tables but 8-bytes in the shadow (3-level) page tables.
- As a consequence, whenever the guest allocates an L1 (lowest level) page table page, the shadow page table code needs to allocate two L1 pages for it.
- Furthermore, the shadow code also needs to manage an additional page table table (L3) which has no direct correspondence in the guest page table.

### 2.1.3 PSE (Page Size Extensions) Support

The PSE flag enables large page sizes: either 4-MByte pages or 2-MByte pages when the PAE flag is set. For 32-bit guests, we simply disable PSE by `cpuid` virtualization. For x86-64 or x86 PAE guests PSE is often a prerequisite: the system may not even boot if the CPU does not have the PSE feature. To address this, we emulate the behavior of PSE in the shadow page tables. The current implementation of 2MB page support is to fragment it into a set of 4KB pages in the shadow page table, since there is no generic 2MB page allocator in Xen.

A set of primitives against the guest and shadow page tables are defined as shadow operations—`shadow_ops`. To avoid code duplication, we currently use a technique whereby we compile the same basic code three times—once each for x86, x86 PAE, and x86-64—but with different implementations of key macros each time. The appropriate `shadow_ops` is set at runtime according to the virtual CPU state.

### 2.1.4 SMP Support

SMP support requires various additional functionality:

- *Local APIC*. To handle IPIs (interprocessor interrupts), SMP guests require the local APIC. The local APIC virtualization has been incorporated in the Xen hypervisor to optimize performance.
- *I/O APIC*. SMP guests also typically require the use of one or more I/O APIC(s). I/O APIC virtualization has been incorporated in the Xen hypervisor for the same reason above.
- *ACPI*. The ACPI MADT table is dynamically set up to specify the number of virtual CPUs. Once ACPI is present, guests expect that the standard or full ACPI features be available. During development this caused a succession of problems, including support for the ACPI timer, event handling, etc.
- *SMP-safe shadow page tables*. At the time of writing the shadow page table code uses a single ‘big lock’ per-domain so as to simplify the implementation. To improve the scalability we need fine-grained locks.

Although HVM SMP guests are stable, we are still working on performance optimizations and scalability. For example, I/O device models need to be multi-threaded to handle simultaneous requests from guests.

### 2.2 Ongoing Work

Ongoing work is looking at optimizing the performance of emulated I/O devices. Unlike paravirtualized guest operating systems, HVM domains are not aware that they are running on top of Xen. Hence they attempt to detect H/W devices and load the appropriate standard device drivers. Xen must then emulate the behaviour of the relevant devices, which has a consequent performance impact.

The general model introduced before Xen 3.0.0 shipped was to use a *device model* assistant process running in Domain 0 which could emulate a set of useful devices. I/O accesses from the unmodified guest would trap, and cause a small *I/O packet* message to be sent to the emulator. The latency involved here can be quite large and so reduces the overall performance of the guest operating system.

A first step to improving performance was to move the emulation of certain simple platform devices into Xen itself. For example, the APIC and PIT are not particularly complicated and, crucially, do not require any interaction with the outside world. These devices tend to be accessed frequently within operating systems and hence emulating these within Xen itself reduces latency and improves performance.

A more substantial set of changes will address both performance and isolation aspects: the *super-emulator*. This approach involves associating a paravirtualized stub-domain with every HVM domain which runs with the same security credentials and within the same resource container. Then when a HVM domain attempts to perform I/O, the trap is instead reflected to the stub-domain which performs the relevant emulation, translating the device requests into the equivalent operations on a paravirtual device interface. Hence a simple IDE controller, for example, can be emulated entirely within the super-emulator but ultimately end up issuing block read/write requests across a standard Xen device channel. As well as providing excellent performance, this also means that HVM domains appear the same as paravirtual domains from the point of view of the tools, thus allowing us to unify and simplify the code.

## 3 MiniXen: A Single Xen Kernel

The purpose of the miniXen project is to take a guest Linux kernel which has been ported to the Xen interfaces and combine this with a thin version of Xen that interacts directly with the native hardware allowing a single domain to run with near native performance. This miniXen performs the bare minimum of operations needed to support a guest OS and where possible passes events, such as interrupts and exceptions, directly to the guest OS omitting the normal Xen protection mechanisms.

An important first step in this was to allow the guest kernel to run in x86 privilege ring zero rather than ring one as a normal Xen guest does. This is easily achieved by using the features flags which Xen exports to all guests: a guest checks for the relevant flag and runs in either ring zero or one as appropriate.

An unfortunate side-effect of running the guest kernel in ring zero is that a privilege level change no longer occurs when making a hypercall or when an interrupt or exception interrupts the guest kernel. This means that the hardware will no longer automatically switch the guest kernel stack for Xen's own stack. Xen relies on its own stack in order to store certain state information. Therefore miniXen must check at each entry point whether the stack pointer points to memory owned by the guest kernel or Xen and if necessary to fix up the stack by locating the Xen stack via the TSS and moving the current stack frame to the Xen stack before continuing.

These checks and the movement of the stack frame necessarily incur a performance penalty which it is desirable to avoid. As all hypercalls were to be either reimplemented or stripped down in order to achieve the goal of performing the bare-minimum of work in miniXen it was possible to also ensure that the hypercalls did not require any state from the Xen stack. Once this was achieved it was possible to turn each hypercall into a simple call instruction by rewriting the miniXen hypercall transfer page, thus avoiding an `int 0x82` hypercall and the expensive stack fixup logic. A direct call into the hypervisor is possible because unlike Xen, miniXen does not need to truncate the guest kernel's segment descriptors in order to isolate the kernel from the hypervisor.

Work is currently on-going to audit miniXen's interrupt and exception handling code to remove any need for state to be stored on Xen's stack and so allow those routines to run on the guest kernel's stack. Once this is complete then miniXen should have no need for a stack of its own and can simply piggy-back on the guest kernels stack except for under very specialized circumstances such as during boot.

Running the guest kernel in ring 0 allows us to once again take advantage of the `sysenter`/`sysexit` instructions to optimize system calls from user space to the guest kernel compared with the normal `int 0x80` mechanism. Normally the `sysenter` instruction is not available to guests running under Xen because the target code segment must be in ring 0. However with the guest kernel running in ring 0 it is simple to arrange for the `sysenter` instruction to jump directly into the guest kernel. The sysenter stack is configured such that it points to the location of the TSS entry containing the guest kernel's stack allowing the kernel to immediately switch to its own stack at the sysenter entry point.

Normally Xen prevents a guest OS from creating a writable mapping of pages which are part of a page table or descriptor table (GDT or LDT) in order to trap any writes and audit them for safety. For this reason guest kernels only create read-only mappings to such pages and a write therefore involves creating an additional writable mapping of a page in the hypervisor's address space. However miniXen does not need to audit the page or descriptor tables and by making use of feature flags exported from Xen to the guest kernel can cause the kernel to create writable mappings of these pages. This allows miniXen to write directly to these pages and therefore allows page table updates with native performance.

## 4 IA64

In the year since the last symposium, Xen/ia64 has made excellent progress and is slowly catching up to Xen on x86-based platforms. The Xen/ia64 community has grown substantially, with major contributions from organizations around the world; and the `xen-ia64-devel` mailing list has grown to over 160 subscribers.

Since Xen's first release on x86, paravirtualization has delivered near-native performance and it is important to demonstrate that these techniques apply to other architectures. Much effort was put into a paravirtualized version of Linux/ia64 and using innovative techniques (e.g. "hyper-registers" and "hyper-privops"), performance was driven to within 2% of native (as measured by a software development benchmark). With guidance from the Linux/ia64 maintainers, an elegant patch was developed that adds a clean abstraction layer to certain privileged instruction macros and replaces only a handful of assembly routines in the Linux/ia64 source. Interestingly, after this patch is applied the resultant Linux/ia64 binary can be run both under Xen and on bare metal—a concept dubbed *transparent paravirtualization*.

Block driver support using Xen's frontend/backend driver model was implemented last summer and integrated into the Xen tree. Soon thereafter Xen/ia64 was supporting multiple Linux domains and work was recently completed to cleanly shutdown domains and reclaim resources. All architectural differences were carefully designed and implemented to fully leverage the Xen user-space tools so that administrators use identical commands on both Xen/x86 and Xen/ia64 for deploying and controlling guest domains.

Preliminary support for Intel Virtualization Technology for Itanium processors (VT-i) was completed last fall and has become quite robust. It is now possible to run unmodified (fully virtualized) domains in parallel with paravirtualized domains. Here also, existing device models and administrative control panel interfaces were leveraged from the VT-x implementation for Xen/x86 to minimize maintenance and maximize compatibility.

In many ways, Xen is an operating system and, as an open-source operating system, there is no need to re-create the wheel. Xen/x86 leveraged a fair amount of code from an earlier version of Linux and periodically integrates code from newer versions. Xen/ia64 goes one step further and utilizes over 100 code modules and header files from Linux/ia64 directly. About half of these files are completely unchanged and the remainder require only minor changes, which, for maintenance purposes, are clearly marked with `ifdefs`. Since Linux/ia64 is relatively immature and subject to frequent bug fixes and tunings, Xen/ia64 can rapidly incorporate these changes.

The value of this direct leverage was demonstrated last fall when SMP host support was added to Xen/ia64. Addition of SMP support to an operating system is often a long painful process, requiring extensive debugging to, for example, isolate and repair overlooked locks. For Xen/ia64, SMP support was added by one developer in a few weeks because so much working SMP code was already present or easily leveraged from Linux/ia64. Indeed, SMP guest support was also recently added in-tree and testing for both SMP host and SMP guest support is showing surprising stability.

While Xen/ia64 has made great progress, much more work lies ahead. Driver domain support, recently added back into Xen/x86, is especially important on the large machines commonly found in most vendors' Itanium product lines. Migration support may prove similarly important. Some functionality has been serialized behind a community decision to fundamentally redesign the Xen/ia64 physical-to-machine mapping mechanisms, which also was a prerequisite for maximal leverage and enablement of the Xen networking split driver code. With the completion of this redesign in late Spring, networking performs well and it is believed that driver domains, as well as domain save/restore and migration will all be easier to implement and will come up quickly.

## 5 PowerPC

Xen is being ported to the PowerPC architecture, specifically the PowerPC 970. The 970 processor contains processor-level extensions to the PowerPC architecture designed to support paravirtualized operating systems. These hardware modifications, made for the hypervisor running on IBM's pSeries servers, allow for minimal kernel changes and very little performance degradation for the guest operating systems. The challenge in a Xen port to PowerPC is fitting the Xen model, developed on desktop-class x86 systems, into this PowerPC architecture.

One of the challenges for Xen on PowerPC isn't related to Xen itself, but rather the availability of hardware platforms capable of running Xen. Although IBM's PowerPC-based servers have the processor hypervisor extensions we are exploiting in Xen, they also contain firmware that doesn't allow user-supplied code to exploit those extensions. Current PowerPC Xen development efforts have been on the Maple 970 evaluation board and the IBM Full System Simulator. The existing firmware on Apple's Power Macintosh G5 systems, which are based on the PowerPC 970, disables the hypervisor extensions completely.

As a secondary task, work is underway to run Xen on PowerPC 970 with hypervisor mode disabled, specifically Apple G5 systems. Although possible, this model requires significant modifications to the guest operating system (pushing it to user privilege mode) and will also incur substantial performance impact. This port will be a stepping stone for supporting all "legacy" PowerPC architectures such as Apple G4 and G3 systems, and embedded processors.

### 5.1 Current Status

On PowerPC, the Xen hypervisor boots from firmware and then boots a Linux kernel in Dom0. The Dom0 kernel has direct access to the hardware of the system, and so device drivers initialize the hardware as in a normal Linux boot. Once Dom0 has booted, a small set of user-land tools are used to load and start Linux kernels in unprivileged domains. At the time of publication, the DomUs have no support for virtual IO drivers, so interaction with the domain isn't currently possible. However, one can see from their boot output that they make it to user-space.

The PowerPC development tree is currently being merged with the main Xen development tree. PowerPC Xen is not yet integrated with Xen's management tools, and the unprivileged domains, lacking device drivers, do not yet perform meaningful work. Once they do, it will also become important to integrate with Xen's testing infrastructure and also to package builds in a more user-friendly manner.

### 5.2 Design Decisions

The PowerPC architecture differs from the x86 in a number of significant ways. In the following we comment on some of the key design decisions we made during the port.

**Hypervisor in Machine Mode** PowerPC, like most RISC-like processors, is able to address all memory while translation is off (i.e., MMU disabled). This allows the hypervisor to execute completely in machine (or "real") mode which removes any impact to the MMU when transitioning from domain to hypervisor and back. Bypassing the MMU allows us to avoid TLB flushes, a significant factor in performance. This decision does have two negative impacts though: it complicates access to MMIO registers and inhibits the use of domain virtual addresses.

The processor must access MMIO registers without using the data cache. This is usually implemented via attributes in MMU translation entries, but since we run without translation this method is unavailable to the hypervisor. Fortunately, there is an architected mode that allows the processor to perform the cache-inhibited load/store operations while translation is off. The problem of accessing memory through a domain virtual address requires a more complex software solution.

Many Xen hypercalls pass the virtual addresses of data structures into the hypervisor. Xen could attempt to translate the virtual addresses to machine addresses without the aid of the MMU, but the MMU translation mechanism of PowerPC is complex enough to make software MMU translation infeasible. An additional complication is that these data structures could span a page boundary (and some span many pages), and although those pages are virtually contiguous, they will likely be discontiguous in the machine address space.

After much agony, the solution developed to overcome this problem is essentially to create a physically addressed scatter/gather list to map the data structures being passed to the hypervisor. Since user-space is unaware of the physical addresses, the kernel must intercept hypercalls originating from user-space and create these scatter/gather structures with physical addresses. The kernel then passes the physical address of the scatter/gather structure to Xen (which is able to trivially convert physical addresses to machine addresses). The end result is Xen's `copy_from_guest()` is passed the address of this special data structure, and copy data from frames scattered throughout the machine address space.

**Hypercalls** PowerPC Linux currently runs on the hypervisor that currently ships with IBM high end products. Like Xen, the POWER Hypervisor virtualizes the processor, memory, interrupts and presents a Virtual IO interface. In order to capitalize on the existing Linux implementation, Xen on PowerPC has adopted the same memory management interfaces as the POWER Hypervisor. However, the support for interrupts and Virtual IO come from the Xen model. The strategy has resulted in a small patch (< 200 LOC) for existing Linux code, and an additional Xen "platform" for `arch/powerpc`, with the rest of the Xen-specific code in the common `drivers/` directory. Since PowerPC Linux supports multiple platforms in the same binary, the same kernel can run on Xen, hardware, or other hypervisors, with no recompile needed.

**Interrupt Model** On most PowerPC processors, interrupts are handled by delivering exceptions to fixed vectors in real-mode, and are differentiated into two classes, synchronous and asynchronous.

Synchronous interrupts are "instruction-caused," which include page faults, system calls, and instruction traps. Under Xen, the processor is run in a mode so that all synchronous interrupts are delivered directly to the domain. The hypervisor extensions provide a special form of system call that is delivered to the hypervisor, which is used for hypercalls.

Asynchronous interrupts are caused by timers and devices. Timer interrupts are also delivered directly to the domain. The hypervisor extensions provide us with an additional timer that is delivered to the hypervisor in order to preempt the active domain.

Device interrupts, called "external exceptions," are delivered directly to the hypervisor, which then creates an event for the appropriate domain. In PowerPC operating systems, the external exception handler probes an interrupt controller to identify the source of the interrupt. In order to deliver the interrupt to the domain, we supplied an interrupt controller driver for Linux that consults Xen's event channel mechanism to determine the pending interrupt.

**Memory Management** The PowerPC hypervisor extensions define a Real Mode Area (RMA), which isolates memory accessed in real mode (without the MMU). This allows for the interrupts that are delivered directly to the domain to be delivered in real mode—the same way they would work without a hypervisor. A side effect of this is that domain "physical" address space must be zero based. However, since the physical address space is now different from the machine address space, supporting DMA becomes problematic. Rather than expose the machine address space to the domain and write a DMA allocator in Linux to exploit it, on 970-based systems we use the I/O Memory Management Unit (IOMMU) that Linux already exploits.

**Atomic Operations** The primary target of Xen are the 32- and 64-bit variants of the x86 architectures. That architecture contains a plethora of atomic memory operations that are normally not present in RISC processors. In particular, PowerPC will only perform atomic operations on 4-byte words (64-bit processors can also perform 8-byte operations), and they must be naturally aligned. This presents a portability issue that must be resolved to support non-x86 architectures.

### 5.3 Conclusion

Xen has created an exceptional virtualization model for an architecture that many consider overly complex and has trailed the industry in virtualization support. Xen's virtualization model developed in the absense of a hardware framework, so the overall challenge of the PowerPC port has been to adapt the Xen model to exploit the capabilities of our hardware.

## 6 Xen Roadmap

Xen continues to develop apace. In this final section we discuss four interesting ongoing or future pieces of work.

### 6.1 NUMA Optimization

NUMA machines, once rarities except on big-iron systems, are becoming more and more the norm with the introduction of multi-core and many-core processors. Hence understanding memory locality and node topology has become even more important.

For Xen, this involves work in at least three areas. Firstly, we need to build a NUMA-aware physical memory allocator for Xen itself. This will enable the allocation of memory from the correct zone or zones and avoid the performance overheads associated with non-local memory accesses.

Secondly, we need to make Xen's CPU scheduler NUMA-aware: in particular it is important to schedule a guest on nodes with local access to the domain's memory as far as is possible. This is complicated on Xen since each guest may have a large number of virtual CPUs (vCPUs) and an opposing tension will be to disperse these to maximize benefit from the underlying hardware.

Finally, the NUMA information really should be propagated all the way to the guest operating system itself, so that a NUMA-aware guest OS can make sensible memory allocation and scheduling decisions for itself. All of this becomes even more challenging as vCPUs may migrate between physical CPUs from time to time.

### 6.2 Supporting IOMMUs

An IOMMU provides an address translation mechanism for I/O memory requests. Popular on big iron machines, they are becoming more and more prevalent on regular x86 boxes—indeed, a primitive IOMMU in the form of the AGP GART has been found on x86 boxes for a number of years. IOMMUs can help avoid problems with legacy devices (e.g., 32-bit-only PCI devices) and can enhance security and reliability by preventing buggy device drivers or devices from performing out-of-bounds memory accesses.

This latter ability is incredibly promising. One reason for the catastrophic effect of driver failure on system stability is the total lack of isolation that pervades device interactions on commodity systems. By wisely using IOMMU-technology in Xen, we hope we shall be able to build fundamentally more robust systems.

Ideally this will not require too much work since Xen's grant-table interfaces were explicitly designed with IOMMUs in mind. In essence, each device driver (virtual or otherwise) can register a page of memory as inbound or outbound for I/O—after Xen has checked the permissions and ownership, an IOMMU entry can be installed allowing the access. After the I/O has completed, the entry can be removed.

### 6.3 Interfacing with 'Smart' Devices

A number of newer hardware devices incorporate additional logic to allow direct access from user-mode processes. Most such devices are targeted toward low-latency zero-copy networking, although storage and graphic devices are moving in the same direction. One interesting piece of future work in Xen will involve leveraging such hardware to enable direct access from guests (initially kernel-mode but ultimately perhaps even from guest user-mode).

As with the above-mentioned work on IOMMUs, this will build on the current grant-table architecture. However additional thought is required to correctly support *temporal* aspects such as the controlled scheduling of user requests. Initial work here is focusing on Infiniband controllers where we hope to be able to provide extreme low-latency while maintaining safety.

### 6.4 Virtual Framebuffer

Past versions of Xen have focused mostly on server oriented environments. In these environments, a virtual serial console that is accessible remotely through a TCP socket or SSH is usually enough for most use cases. As Xen expands its user base and begins to be used in other types of environments, a more advanced guest interface is required. The Xen 3.0.x series will introduce the first of a series of features designed to target these new environments starting with a paravirtual framebuffer.

A paravirtual framebuffer provides a virtual graphics adapter that can be used to run graphical distribution installers, console mode with virtual terminal switching and scrollback, and windowing environments such as the X Window System. The current implementation allows these applications to run with no modifications and no special userspace drivers. Future versions will additionally provide special interfaces to userspace so that custom drivers can be written for additional performance (for instance, a custom X.org driver).

Traditional virtualization systems such as QEmu, Bochs, or VMware provide graphics support by emulating an actual VGA device.

This requires a rather large amount of emulation code since VGA devices tend to be rather complex. VGA emulation is difficult to optimize as it often requires MMIO emulation for many performance critical operations such as blitting. Many full virtualization systems use hybrid drivers featuring additional hardware features that provide virtualization specific optimized graphics modes to avoid MMIO emulation.

In contrast, a fully paravirtual graphics driver offers all of the performance advantages of a hybrid driver with only a small fraction of the amount of code. Emulation for the Cirrus Logic chipset provided by QEmu requires over 6,000 lines of code (not including the VGA Bios code). The current Xen paravirtual framebuffer is implemented in less than 500 lines of code and supports arbitrary graphic resolutions. The QEmu graphics driver is limited to resolutions up to 1024x768.

The paravirtual framebuffer is currently implemented for Linux guests although the expectation is that it will be relatively easy to port to other guest OSes. The driver reserves a portion of the guests memory for use as a framebuffer. The location of this memory is communicated to the framebuffer client. The framebuffer client is an application, usually running in the administrative domain, that is responsible for either rendering the guest framebuffer to the host graphics system or over the network using a protocol such as VNC. The client can directly map the paravirtual framebuffer's memory using the Xen memory sharing APIs.

An initial optimization we have implemented uses the guest's MMU to reduce the amount of client updates. Normally, applications within a guest expect to be able to write directly to the framebuffer memory without needing to provide any sort of flushing or update information. This presents a problem for the client since it has no way of knowing which portions of the framebuffer is updated during a given time period. We are able to mitigate this by using a timer to periodically invalidate all guest mappings of the framebuffer's memory. We can then keep track of which pages were mapped in during this interval. Based on the dirtied page information, we can calculate the framebuffer region that was dirtied.

In practice, this optimization provides updates at a scanline granularity. In the future, we plan on enhancing the guest's userspace rendering applications to provide greater granularity in updates. This is particularly important for bandwidth sensitive clients (such as a VNC client).

We also plan on exploring other graphics related features such as 2D acceleration (region copy, cursor offloading, etc) and 3D support. There are also some interesting security related features to explore although that work is just beginning to take shape. Future versions of Xen may also provide support for other desktop-related virtual hardware such as remote sound.

## 7 Conclusion

Xen continues to evolve. Although it already provides high performance paravirtualization, we are working on optimizing full virtualization to better serve those who cannot modify the source code of their operating system. To simplify system administration, we are working on supporting a single linux kernel binary which can run either directly on the hardware or on top of Xen. To allow a broader applicability, we are enhancing or developing support for non x86 architectures. And we are looking beyond these to develop new features and hence to ensure that Xen remains the world's best open source hypervisor.