Safe Hardware Access with the Xen Virtual Machine Monitor

Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson**
University of Cambridge Computer Laboratory, J J Thomson Avenue, Cambridge, UK
{firstname.lastname}@cl.cam.ac.uk

Abstract

The Xen virtual machine monitor allows multiple operating systems to execute concurrently on commodity x86 hardware, providing a solution for server consolidation and utility computing. In our initial design, Xen itself contained device-driver code and provided safe shared virtual device access. In this paper we present our new *Safe Hardware Interface*, an isolation architecture used within the latest release of Xen which allows unmodified device drivers to be shared across isolated operating system instances, while protecting individual OSs, and the system as a whole, from driver failure.

1 Introduction

We have recently developed Xen [1], an x86-based virtual machine manager specifically targeting two utility-based computing environments:

1. organizational compute/data centers, where a large cluster of physical machines may be shared across different administrative units; and
2. global-scale compute utilities, such as our own XenoServers [2] initiative, in which completely unrelated customers may lease a set of distributed resources to deploy their own services.

In both of these situations, Xen must provide reliable execution of OS instances, hard isolation, and accounting and management for the underlying physical resources.

This paper focuses on our work addressing dependability on commodity hardware by isolating device driver code yet retaining the ability to share devices across OS instances. Device drivers are well known to be a major source of bugs and system failures, and the *sharing* of devices raises the stakes of driver dependability drastically. The wide variety of hardware available for PCs and the desire to share device access has led us to an architecture in which we execute unmodified device drivers in isolated "driver domains": in essence, driver-specific virtual machines.

*Intel Research Cambridge, UK

To achieve this robustly, we have developed a *Safe Hardware Interface* which allows the containment of practically all driver failures by limiting the driver's access to the specific hardware resources (memory, interrupts, and I/O ports) necessary for its operation. Our model, which places device drivers in separate virtual OS instances, provides two principal benefits: First, drivers are *isolated* from the rest of the system; they may crash or be intentionally restarted with minimal impact on running OSes. Second, a *unified interface* to each driver means that drivers may be safely shared across many OSes at once, and that only a single low-level driver implementation is required to support different paravirtualized operating systems.

While general approaches to partitioning and protecting systems for reliability have been explored over the past thirty years, they often depend on specific hardware support [3], or are unconcerned with enterprise-class performance and dependability [4, 5]. Our work addresses the problem of ensuring the reliability of shared device drivers for enterprise services on the PC architecture without requiring specialized hardware support.

2 Related Work

The current PC I/O architecture presents a multifaceted set of challenging problems. This section attempts to summarize the great breadth of previous work that has attempted to tackle individual aspects of the problem. We have drawn on many of these efforts in our own research. There are two broad classes of work related to our own. First is a large set of efforts both in systems software and hardware development toward safe isolation. Second are attempts to better structure the interfaces between devices and their software, and the OSs and applications they interact with.

2.1 Safe Isolation

Researchers have long been concerned with the inclusion of extension code in operating systems. Extensible operating systems [6, 7] explored a broad range of approaches

to support the incorporation of foreign, possibly untrusted code in an existing OS. Swift et al [4] leverage the experiences of extensibility, particularly that of interposition, to improve the reliability of Linux device drivers. While their work claims an improvement in system reliability it demonstrates the risk of a narrow focus: their approach sacrifices performance drastically in an attempt to add dependability without modifying the existing OS. By addressing the larger architectural problem and not fixating on a single OS instance, we provide higher performance and solve a broader set of issues, while still remaining compatible with existing systems.

Our implementation, presented in Section 4, uses a virtualization layer to achieve isolation between drivers and the OS (or OSs) that use them. Providing a low-level systems layer that is principally responsible for managing devices was initially explored in Nemesis [8] and the Exokernel [9]. Our work refines these approaches by applying them to existing systems. Additionally, Whitaker et al [10] speculate as to the potential uses of a virtualized approach to system composition, drawing strongly on early microkernel efforts in Mach [11] among others [12, 13]. Our work represents a realization of these ideas, demonstrating that isolation can be provided with a surprisingly low performance overhead.

Commercial offerings for virtualization, such as VMware ESX Server [14], allow separate OSs to share devices. While we have previously demonstrated [1] that our approach to virtualization provides higher performance, this work moves to focus specifically on additional concerns such as driver dependability; our implementation is now not only faster but also accommodates a strictly higher level of driver dependability.

Several research efforts have investigated hardware-assisted approaches to providing isolation on the PC platform. The Recovery-Oriented Computing [15] project, whose goals are similar to our own, have used hardware for system diagnostics [16], but defer to 'standard mechanisms' for isolation. Intel's SoftSDV [17], which is a development environment for operating systems supporting the IA-64 instruction set, uses PCI riser cards to proxy I/O requests. While their concern is in mapping device interrupts and DMA into the simulated 64-bit environment, the same approach could be used to provide device isolation. Intel has also announced that their new LaGrande architecture [18] will protect memory from device DMA.

2.2 Better Interfaces

Our goal of providing more rigid OS–device interfaces is hardly new. Most notably, corporate efforts such as UDI [19] have attempted to do just this. There are two key limitations of UDI that we directly address. Firstly, we enforce isolation whereas UDI-compliant drivers still execute in the same protection domain as the operating system, and thus there is no mitigation of the risks posed by erroneous drivers. Secondly, our external perspective avoids the trap to which vendor consortiums such as UDI often fall victim: that of 'interface unioning'. Rather than providing the aggregate interface present in all existing drivers, we settle on a narrower, *idealized* interface. While we provide mechanisms to directly (and safely) expose the hardware should our interface be too constrictive, we have not found this to be a problem in our experiences with a large number of network and storage devices—most relevant for server-class machines—and several OSs.

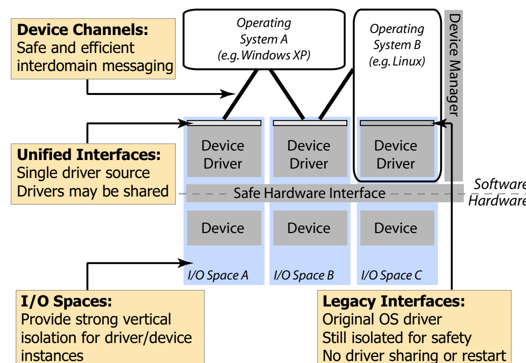

Figure 1: Design of the safe hardware model.

Novel OS architectures have long struggled with a lack of device driver support. The vast number of available devices has compounded this problem, making the adoption of an existing driver interface attractive for fledgling systems. Microkernel systems such as Fluke [20] and L4 [21] have investigated wrapping Linux device drivers in customized interfaces [22, 23]. Although the structure of our architecture is not entirely unlike a microkernel, our intent is to solve the driver interface issue for all operating systems on the PC architecture, rather than make a small set of existing drivers work for a single experimental OS.

Perhaps most closely related to our work are recent attempts by researchers to use a microkernel approach to allow the *reuse* of unmodified device drivers within newer operating systems [24, 25]. We are less concerned with using legacy drivers in modern operating systems than with providing shared access to isolated device drivers.

3 Design

This section presents the high-level design issues that have driven our work. We begin by discussing the issues involved in achieving *isolation* between virtualized OS instances, device driver code, and physical devices. In the second half of this section, we go on to discuss design concerns for unified interfaces.

As illustrated by Figure 1, our architecture comprises three parts. Firstly, we introduce *I/O Spaces* which arrange that devices perform their work in *isolation* from the rest of the

Requirement 1: **Driver Isolation**	
Memory:	execute in logical fault domain
CPU:	schedule to prevent excessive consumption
Privilege:	limit access to instruction set
Requirement 2: **Driver → Device Isolation**	
I/O Registers:	restrict access to permitted ranges
Interrupts:	allow to mask/receive only device's interrupt
Requirement 3: **Device Isolation**	
Memory:	prevent DMA to arbitrary host memory
Other Devices:	prevent access to arbitrary other devices

Table 1: Requirements for Safe Hardware

system. This increases reliability by restricting the possible harm inflicted by device faults. Secondly, we define a set of per-class *unified interfaces* that are implemented by all devices of a particular type. This provides driver portability, avoiding the need to reimplement identical functionality for a range of different OS interfaces. Finally, our device manager provides a consistent *control and management* interface for all devices, simplifying system configuration and diagnosis and treatment of device problems.

3.1 Isolation

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. The issues that must be addressed to achieve full isolation are outlined in Table 1. The concerns are divided into three requirements: isolating the execution of driver code from other software components, ensuring that drivers may only access the device they manage, and enforcing safe device behavior.

Previous attempts at driver isolation [4] have placed driver code in a separate logical fault domain, essentially providing virtual memory protection between the driver and the rest of the system. However, this is only a partial solution as it primarily protects memory; a logical isolation layer *must* be used to provide isolation of scheduling and access to privileged instructions.

The implementation that we present in Section 4 uses a virtual machine monitor (VMM) to achieve the required logical isolation between driver and OS code, as identified by Requirement 1 in Table 1. By tracking and retaining full control of each driver's CPU and memory use, the VMM provides isolation guarantees analogous to an OS and its application processes. For example, if a faulty driver attempts to write to a memory location outside its heap, the damage is contained to the VM that the driver is executing within. Furthermore, by containing driver and OS in an isolated virtual machine, misbehaving drivers may be restarted with a minimal impact on the rest of the system.

The complete isolation of device access is a fundamental problem on the x86 architecture, which provides no specific hardware support to limit access to specific devices, or to limit device access to system memory. Over the course of our design and implementation, we have developed the notion of an *I/O Space* to describe the underlying mechanism required to achieve complete isolation. An I/O Space is a vertical protection domain that can be assigned a set of physical resources for a specific device and driver interaction, including memory, device registers, and interrupts. The intent of an I/O Space is to make the set of accesses between a driver (or ideally even a specific client) and a physical device a first-class entity on the system.

A complete realization of our I/O Spaces would require chipset support, in that they would address the concerns outlined in Requirement 3 above. Providing protection against malicious or erroneous device DMA, and arbitrating device access to shared buses simply cannot be achieved without hardware support.

However, as the implementation that we present in Section 4 uses virtualization, we have been able to address the physical isolation problems of host-to-device access, Requirement 2 above, by implementing within the VMM the I/O-Space functionality of a next-generation chipset. We believe that the isolation we have achieved is the strongest possible without hardware modifications. Although our current implementation cannot protect against unsafe device DMA, we describe the minor modifications that would be necessary to take advantage of a safe DMA controller. Emerging hardware research [16, 17, 18] indicates that these hardware improvements may soon be incorporated into the PC platform.

3.2 Unified Interfaces

Although the PC has standardized hardware interfaces there is no such accepted standard for the interface to system software, despite industry efforts [19]. Our solution is to define a set of idealized high-level interfaces tailored for each class of device. OS vendors then need implement only a single, small driver per *device class* that communicates via the unified interface: this can be developed in-house by developers with intimate knowledge of the OS, and subjected to appropriate quality-control checks. By implementing the unified interface, hardware vendors automatically support every PC system. Furthermore, they may arbitrarily choose how the implementation is divided between hardware and software, perhaps incorporating more functionality into higher-cost products that include advanced features such as I/O processors [26].

Our unified device interfaces are based on those provided by the Xen VMM which, as we have previously demonstrated [1], provides low-overhead access to common device classes. The essential features required for efficient data-path communication are to avoid data copies, to provide back pressure to the data source, and to use a flexible and asynchronous notification primitive. Within our architecture we incorporate these principles into *device*

channels, linking the unified interfaces exported by device drivers to the operating systems using them. We provide details of a software implementation of device channels in Section 5. However, we are careful in our design not to exclude the possibility of a hardware implementation.

Concerns regarding the feasibility of adopting standardized device interfaces are very relevant, as acceptance is more of a political problem than a technical one. Our efforts to date have had a great deal of success in allowing a variety of networking and storage devices to function through a common interface to Linux, NetBSD, and Windows XP. We have focused on these classes of device as we believe that network and disk are the two most crucial device interfaces in a server environment. We do not presume that the interfaces we have identified are complete, and expect them to evolve over time. However, experience so far has shown that our model is valid; other groups (e.g. [27]) have independently ported new devices to our architecture with minimal effort.

While we believe that unified interfaces provide considerable benefit, we must also acknowledge that it is likely impossible to effectively model all devices: emerging devices and special-purpose applications must be considered. In these situations, we allow device access to be exposed directly, and it is through this mechanism that we address video and sound devices in our current implementation. Note that even when we do not use a unified virtualized device interface, the architecture still provides isolation and safety. This transitional approach allows our architectural benefits to be realized in the short term, while we move to focus on the challenging problems of sound and video interfaces in the future.

It is additionally worth observing that organizations continue to move toward OS virtualization as a means of making better use of server hardware. Unified interfaces are particularly advantageous in a virtualized environment where they can enable *device sharing*.

An example of unified interfaces, legacy support, and device sharing was shown in Figure 1 in which two operating systems and three device drivers all run on a single machine. The two leftmost device drivers present a unified interface which ‘wraps’ existing driver code. Using this interface means that device drivers may be individually scheduled, shared between operating systems, and restarted in case of error. The rightmost operating system contains a legacy driver; although this prevents separate scheduling or sharing, the safe hardware interface can still be used to limit the driver’s privileges.

3.3 Control and Management

The final concern addressed by our architecture is that of device control and configuration — an area that has been particularly neglected during the PC’s evolution. The lack of standardized platform-wide control interfaces has led to the implementation of unique and proprietary configuration interfaces for each OS and device[1]. A significant disadvantage of this ad hoc approach is that system administrators require additional training for each OS environment and machine setup that they support, simply to understand multiple different configuration interfaces that ultimately provide identical functionality.

The transition of the PC platform into the server room means that manageability is now more important than ever. The current jumble of configuration tools is inappropriate for configuring and managing the large-scale clusters that are common in enterprise environments. Console-based interfaces, although suitable for configuring small numbers of desktop machines, are a major hindrance when configuration changes must be applied to hundreds of machines at a time. The growing problem of remote management is a primary motivation for the LinuxBIOS project [28].

This final aspect of our architecture is handled by a *device manager* — essentially an extension to the system BIOS that provides a common set of management interfaces for all devices. The device manager is responsible for bootstrapping isolated device drivers, announcing device availability to OSs, and exporting configuration and control interfaces to either a local OS or to a remote manager.

4 I/O Spaces

Our safe hardware interface enforces isolation of device drivers by restricting the hardware resources that they can access: we call such a restricted environment an *I/O Space*. To achieve this, 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*, allowing the system controller unfettered access so that it can determine each device’s resources, while restricting each driver’s view of the system so that it cannot see resources that it cannot access.

4.1 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 exam-

[1] Some common device classes do enjoy a consistent control interface, but even this consistency is not carried across different OSs.

ple, 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.

4.2 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 [29].

4.3 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

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.

4.4 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.

5 Device Channels

Although the safe hardware interface can be configured to allow a guest OS to run its own device drivers, this misses potential improvements in reliability, maintainability and manageability. We therefore prefer to run each device driver within an isolated driver domain (IDD) which limits the impact of driver faults.

Guest OSs access devices via *device channel* links with 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 the system controller to introduce an IDD to a guest OS, and vice versa. To facilitate this, the system controller automatically establishes an initial control channel with each domain that it creates. Figure 2 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 system controller. For this purpose, Xen permits restricted *sharing* of memory pages between domains.

5.1 Sharing Memory

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

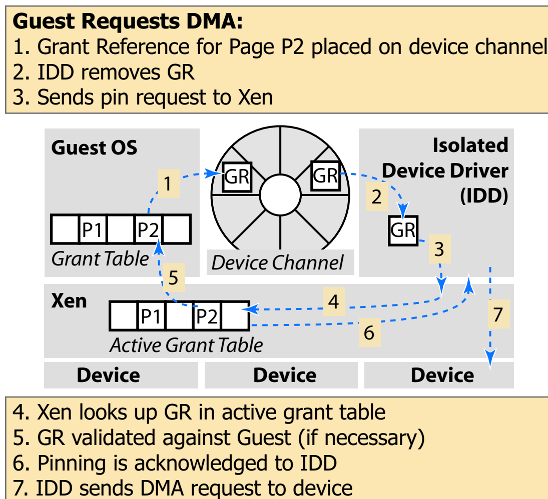

Figure 2: Using device channel to request a data transfer.

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 the guest'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.

5.2 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 4.2. 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.

5.3 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. As described in Section 5.1, 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 2, 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 4.3), 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).

5.4 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 [30], 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.

6 Starting and Restarting IDDs

Responsibility for device management resides with the system controller: a small privileged management kernel that is loaded from firmware when the system boots. During bootstrap, the device manager probes device hardware and creates an IDD, loaded with the appropriate driver, for each detected device. The controller's ongoing responsibilities include per-guest device configuration, managing setup of device channels, providing interfaces for hardware configuration, and reacting to driver failure.

There are several ways in which the controller may determine that a driver has failed: for example, it may receive notification from Xen that the IDD has crashed, or a wedged IDD may fail to unpin guest buffers within a specified time period. The subsequent recovery phase is greatly simplified by the componentized design of our I/O architecture: the shared state associated with a device channel is small and well-defined; and IDD-internal state is 'soft' and therefore may simply be reinitialized when it restarts.

The recovery phase comprises several stages. First, the controller destroys the offending IDD and replaces it with a freshly-initialized instance. The controller then signals to connected guest OSs that the IDD has restarted; each guest is then responsible for connecting itself to the new device channel. At this point, the guest may also reissue requests that may have been affected by the failure (i.e., outstanding requests for which no response has been received). Note that in the case of more sophisticated "stateful" devices in may be in addition necessary to reset the device to a known state so as to ensure that all resources are released and that all reissued requests act as if idempotent.

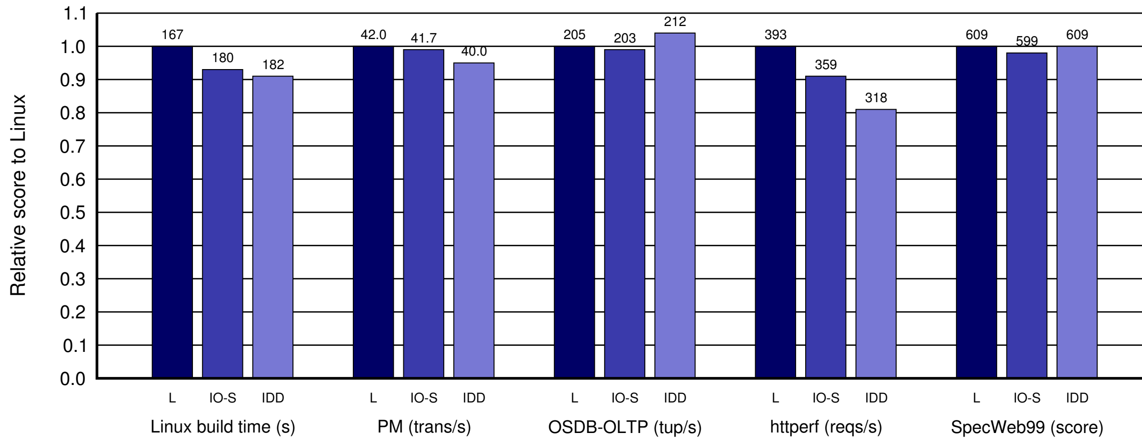

Figure 3: Application-Level Benchmarks. (L=L-SMP, IO-S=IO-Space)

7 Evaluation

In this section we begin by evaluating the impact of our isolation mechanisms on realistic application workloads using industry standard benchmarks such as Postmark, SPEC WEB99 and OSDB. We next investigate the overhead of using the safe hardware interface on network and disk systems, and finally we provoke a series of device-driver failures and measure system availability during recovery.

All experiments were performed on a Dell PowerEdge 2650 dual processor 3.06Ghz Intel Xeon server with 1GB of RAM, two Broadcom Tigon 3 Gigabit Ethernet network cards, and an Adaptec AIC-7899 Ultra160 SCSI controller with two Fujitsu MAP3735NC 73GB 10K RPM SCSI disks. Linux version 2.4.26 and RedHat 9.0 Linux were used throughout, installed on an ext3 file-system. Identical device driver source code from Linux 2.4.26 is used in all experiments, allowing us to measure only performance variations caused by varying the I/O system configuration.

We compare the performance of our IDD prototype against a number of other configurations, using a vanilla Linux 2.4.26 SMP kernel as our baseline (`L-SMP`). To establish the overhead of implementing protected hardware access we measure a version of Xen/Linux containing disk and network drivers that access the hardware via the protected interface (`IO-Space`). We also evaluate the performance of our full-blown architecture using IDDs for each of the network and disk devices, communicating with an instance of Xen/Linux using device-channel I/O interfaces (`IDD`). Each IDD and Xen/Linux instance runs in its own isolated Xen domain on a separate physical CPU.

7.1 Application-Level Benchmarks

We subjected our test systems to a battery of application-level benchmarks, the results of which are displayed in Figure 3. Our first benchmark measures the elapsed time to do a complete build of the default configuration of a Linux kernel tree stored on the local ext3 file system. The kernel compile performs a moderate amount of disk I/O as well as spending time in the OS kernel for process and memory management, which typically introduces some additional overhead when performed inside a virtual machine. The results show that the I/O Space virtualized hardware interface incurs a penalty of around 7%, whereas the full IDD architecture exhibits a 9% overhead.

Postmark is a file system benchmark developed by Network Appliance which emulates the workload of a mail server. It initially creates a set of files with varying sizes (2000 files with sizes ranging from 500B to 1MB) and then performs a number of transactions (10000 in our configuration). Each transaction comprises a variety of operations including file creation, deletion, and appending-write. During each run over 7GB of data is transferred to and from the disk. The additional overhead incurred by I/O Spaces and the full IDD architecture are just 1% and 5% respectively.

OSDB-OLTP is an Open Source database benchmark that we use in conjunction with PostgreSQL 7.3.2. The benchmark creates and populates a database and then both queries and updates tuples in the database. As the default dataset of 40MB fits entirely into the buffer cache, we created a dataset containing one million tuples per relation, resulting in a 400MB database. We investigate the surprisingly high result achieved by IDD in Section 7.3.

httperf-0.8 was used to generate requests to an Apache 2.0.40 server to retrieve a single 64kB static HTML document. The benchmark was configured to maintain a single outstanding HTTP request, thus effectively measuring the response time of the server. The resulting network bandwidth generated by the server is around 200Mb/s. The I/O Space result exposes the overhead of virtualizing interrupts in this latency-sensitive scenario in which there is no opportunity to amortise the overhead by pipelining requests. Communicating with the IDD via the device channel interface compounds the effect by requiring a significant number of inter-domain notifications. Despite this, the response time is within 19% of that achieved by native L-SMP.

	TCP MTU 1500		TCP MTU 552	
	TX	RX	TX	RX
L-SMP	897	897	808	808
I/O Space	897 (0%)	898 (0%)	718 (-11%)	769 (-5%)
IDD	897 (0%)	898 (0%)	778 (-3%)	663 (-18%)

Table 2: *ttcp*: Bandwidth in Mb/s

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. During the measurement period there is up to 200Mb/s of TCP network traffic and considerable disk read-write activity on a 2.7GB dataset. Under this demanding workload we find that the overhead of I/O Spaces and even full device driver isolation to be minimal: just 1% and 2% respectively.

7.2 Network performance

We evaluated the network performance of our test configurations by using *ttcp* to measure TCP throughput over Gigabit Ethernet to a second host running `L-SMP`. Both hosts were configured with a socket buffer size of 128KB as this is recommended practice for Gigabit networks. We repeated the experiment using two different MTU sizes, the default Ethernet MTU of 1500 bytes, and a smaller MTU of 552 bytes. The latter was picked as it is commonly used by dial-up PPP clients, and puts significantly higher stress on the I/O system due to the higher packet rates generated (190,000 packets a second at 800Mb/s).

Using a 1500 byte MTU all configurations achieve within a few percent of the maximum throughput of the Gigabit Ethernet card (Table 2). The 552 byte MTU provides a more demanding test, exposing the different per-packet CPU overheads between the configurations. The virtualized interrupt dispatch used by I/O Spaces incurs an overhead of 11% on transmit and 5% on receive. Hence safe control of interrupt dispatch and device access can be achieved at reasonable cost even under high load.

7.3 Disk performance

Unlike networking, disk I/O typically does not impose a significant strain on the CPU because data is typically transferred in larger units and with less per-operation overhead. We performed experiments using *dd* to repeatedly write and then read a 4 GB file to and from the same ext3 file system (Table 3). Read performance is nearly identical in all cases, but attempts to measure write performance are hampered due to an oscillatory behaviour of the Linux 2.4 memory system when doing bulk writes. This leads to our IDD configurations actually outperforming standard Linux

	read	write
L-SMP	66.01	47.36
I/O Space	65.78(-0%)	46.74(-1%)
IDD	65.16(-1%)	58.47(+23%)

Table 3: *dd*: Bandwidth in MB/s

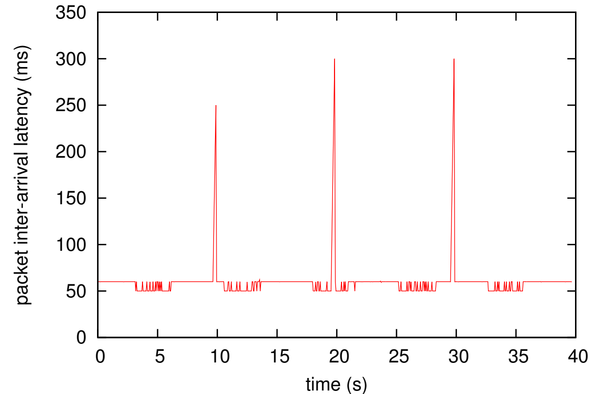

Figure 4: Effect of driver restart on packet arrivals.

as the extra stage of queueing provided by the device channel interface leads to more stable throughput.

7.4 Device Driver Recovery

In these tests we provoked our network driver to perform an illegal memory access, and then measured the effect on system performance. In this scenario detection of the device driver failure is immediate, unlike internal deadlock or infinite looping where there will be a detection delay dependent on system timeouts.

To test driver recovery we caused an external machine to send equally-spaced ping requests to our test system at a rate of 200 packets per second. Figure 4 shows the inter-arrival latencies of these packets at a guest OS as we inject a failure into the network driver domain at 10-second intervals. During the recovery period after each failure we recorded network outages of around 275ms. Most of this time is spent executing the device driver's media detection routines while determining the link status.

8 Conclusion

The safe hardware interface used by Xen places drivers in isolated virtual machines, allowing the use of existing driver code in enterprise computing environments where dependability is paramount. Furthermore, a single driver implementation may be instantiated once yet shared by a number of OSs across a common interface.

Although the hardware required to fully enable our I/O architecture is not yet available, we currently support nearly all of the required features (a notable exception being protection against erroneous DMA). We achieve surprisingly

good performance—overhead is generally less than a few percent—and restartability can be achieved within a few hundred milliseconds. Furthermore, we believe that our implementation can naturally incorporate and benefit from emerging hardware support for protection.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM SOSP*, pages 164–177, October 2003.

[2] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform. In *Proceedings of the 6th OPENARCH*, April 2003.

[3] T. Borden, J. Hennessy, and J.Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, 1989.

[4] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM SOSP*, pages 207–222, October 2003.

[5] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, pages 1–14, 2001.

[6] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, October 1996.

[7] B. Bershad, S. Savage, P. Pardyak, E. Gün 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 Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[8] 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, September 1996.

[9] D. Engler, Kaashoek F, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[10] A. Whitaker, R. Cox, M. Shaw, and S. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 169–182, March 2004.

[11] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications. *IEEE Software*, 2(4).

[12] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126. USENIX Assoc., 1992.

[13] F. Armand. Give a process to your drivers. In *Proceedings of the EurOpen Autumn 1991 Conference*, Budapest, 1991.

[14] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.

[15] A. Brown and D. Patterson. Embracing failure: A case for Recovery-Oriented Computing (ROC). In *Proceedings of the 2001 High Performance Transaction Processing Symposium, Asilomar, CA*, October 2001.

[16] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D.A. Patterson, and K. Yelick. Roc-1: Hardware support for recovery-oriented computing. In *IEEE Transactions on Computers, vol. 51, no. 2*, February 2002.

[17] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A pre-silicon software development environment for the IA-64 architecture. *Intel Technology Journal*, 3(Q4):14, November 1999.

[18] Intel Corp. Lagrande technology architectural overview, September 2003. Order number 252491-001, *http://www.intel.com/technology/security/downloads/LT_Arch_Overview.pdf*.

[19] Introduction to UDI version 1.0. Project UDI, 1999. Technical white paper, *http://www.projectudi.org/*.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, October 1996.

[21] J. Liedtke. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, December 1995.

[22] K. T. Van Maren. The Fluke device driver framework. Master's thesis, University of Utah, December 1999.

[23] C. Helmuth. Generische portierung von linux-gertetreibern auf die drops-architektur, July 2001. Diploma Thesis, Technical University of Dresden.

[24] Joshua LeVassuer and Volkmar Uhlig. A Sledgehammer Approach to Reuse of Legacy Device Drivers. In *Proceeedings of the 11th ACM SIGOPS European Workshop, Leuven, Belgium*, September 2004.

[25] Joshua LeVassuer and Volkmar Uhlig and Jan Stoess and Stefan Goetz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, December 2004.

[26] Intelligent I/O (I_2O) architecture specification, Revision 2.0, 1999. I_2O Special Interest Group.

[27] B. Clark, T. Deshane, E. Dow, S Evanchik, M. Finlayson, J. Herne, and J.N. Matthews. Xen and the art of repeated research. In *Proceedings of the Usenix annual technical conference, Freenix track*, July 2004.

[28] R. Minnich, J. Hendricks, and D. Webster. The Linux BIOS. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

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

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