Reconstructing I/O

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

We present a next-generation architecture that addresses problems of dependability, maintainability, and manageability of I/O devices and their software drivers on the PC platform. Our architecture resolves both hardware and software issues, exploiting emerging hardware features to improve device safety. Our high-performance implementation, based on the Xen virtual machine monitor, provides an immediate transition opportunity for today's systems.

1 Introduction

Device drivers are one of the most troublesome aspects of commodity operating systems — a weakness that has received scant attention as the PC platform has evolved, driven by the overriding goal of affordable performance. It is now no longer sufficient to merely provide ever-improving data speeds; as these systems are increasingly used in business-critical applications, it is essential to review their management and dependability

Driver code is written for, and runs within a specific OS. For purposes of execution, the two are inseparable on modern systems. This entanglement leads directly to three problems that urgently need addressing given the growing stake PCs hold in today's enterprise server environments:

1. **Dependability:** The lack of isolation between execution of driver and OS code sacrifices dependability. Driver errors often cause catastrophic system crashes.
2. **Maintainability:** Device drivers must be rewritten for each OS, and driver code is difficult and expensive to develop and maintain.
3. **Manageability:** The troubleshooting and resolution of driver-related problems is often difficult and time consuming in the ongoing administration of a system.

These concerns reflect the roots of the PC architecture as a desktop platform. Although inconvenient, system crashes and tedious diagnosis of hardware problems are accepted as part of the PC experience by workstation users and administrators. However, in recent years the PC has supplanted mainframes and special-purpose operating systems in the enterprise. When every second of downtime impacts revenue, the traditional limitations of the architecture can no longer be tolerated.

This paper presents a set of changes intended to transition toward dependable, maintainable and manageable systems. We present a new system architecture which addresses fundamentally unsafe issues in the current I/O model, such as unrestricted device DMA. Our architecture is built upon three key ideas. First, we mitigate the risks of device misbehaviour by enforcing isolation between device-granularity protection domains. Second, we introduce a set of simple, unified interfaces between OS and driver software. These interfaces provide the required separation of concerns between these hitherto conflated aspects of systems software. Finally, we unify the control and configuration of devices in a single OS-agnostic system interface.

Our approach is not merely an academic proposal. We have taken advantage of our experience with high performance virtualization techniques to construct a complete implementation of this proposal on existing hardware. Our prototype solves all the above problems, except where constrained by the shortfalls of existing hardware, and provides a transition path toward a new system architecture as well as immediate benefit to present-day systems. In the following subsections we discuss each of the fundamental concerns more specifically.

1.1 Dependability

Drivers on the PC architecture are frequently blamed as a leading cause of system instability [1]. The fact that driver code runs with the same privilege and in the same address space as the operating system means that even simple pointer errors may compromise system stability. However,

[*]Intel Research Cambridge, UK

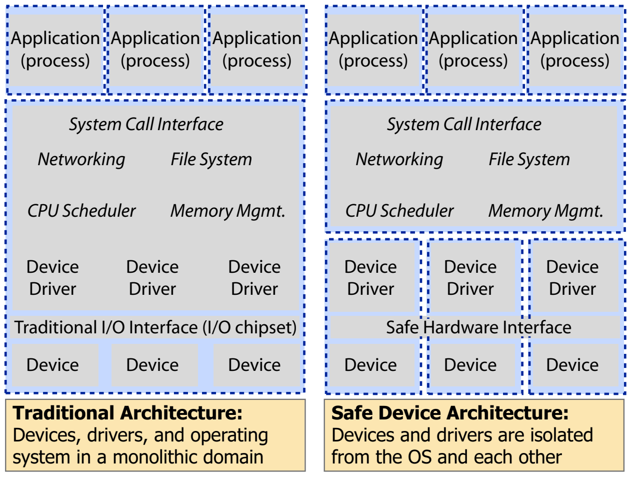

Figure 1: Isolation in a traditional OS (*lhs*) versus the Safe Device approach (*rhs*) which isolates device drivers from the kernel and from each other, and uses a *safe hardware interface* to extend this isolation to the device level.

driver misbehaviour is not confined to just pointer errors: drivers may leak memory, deadlock, fail to correctly manage interrupts, or wedge the system inside an infinite loop.

The broad spectrum of failure possibilities means that making systems dependable requires complete isolation of driver execution from that of the OS, applications and other drivers. More importantly, it is crucial that mechanisms be provided to ensure ongoing device availability, by recognising and reacting to driver failures. Our approach is to enforce *complete vertical isolation* of resources including device hardware, driver code, and operating system.

The difference between this new approach and traditional OS structure is shown in Figure 1 — device drivers are executed in a protection domain which restricts their access to host memory, I/O instructions, device registers and interrupt lines. Section 3 discusses our use of I/O Spaces, the mechanism by which we achieve these vertically isolated slices through the system. In Section 4 we present a set of techniques to address the difficult problem of *recognising* driver failure, and go on to identify problems that simply cannot be solved with current hardware.

1.2 Maintainability

Although hardly a simple undertaking, isolating driver code for safety only addresses a symptom of a considerably broader, architectural problem. Today, the development of driver code is tightly coupled with the individual target operating system. Device vendors must maintain separate source trees for each OS that they hope to support. Moreover, the individual driver OS interfaces are incredibly complex; Swift et al identify more than four hundred interface points between the Linux kernel and driver code [2]. This, combined with the fact that OS interfaces differ not only syntactically but also semantically (e.g., providing very different threading models and communication primitives) means that each driver requires developers with OS-specific expertise.

This coupling benefits no one, except as an inertial force behind established operating systems. Even Windows is victim to the interface problem: Microsoft recently announced that the move to a 64-bit OS will require a new version of every device driver [3]. The lack of a common driver–OS interface is also an inhibitor to new OSs.

Our approach is to unify the driver–OS interface and use the isolation techniques discussed above to execute driver code in a completely separate execution context from that of the OS. The OS is presented with an idealized interface that describes a class of hardware, for instance storage devices or network interfaces. This approach allows a single driver to be used under any number of operating systems – each OS-specific driver serves an entire class of devices and typically comprises just a few hundred lines of code.

1.3 Manageability

Unified access to drivers is not just needed from a data access perspective, but also for administration. Presently, the tasks of diagnosing and configuring hardware are specific to each driver and OS instance. This has led, for instance, to specific device functionality being exposed under Windows but not Linux, and vice versa. The ad hoc means by which devices are currently administered equates to wasted administrator time and all the costs that entails. By unifying the control of devices in addition to their access, we hope to address these administrative concerns.

2 Related Work

The current 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 [4, 5] explored a broad range of approaches to support the incorporation of foreign, possibly untrusted code in an existing OS. Swift et al [2] leverage the experiences of extensibility, particularily 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 principly responsible for managing devices was initially explored in Nemesis [6] and the Exokernel [7]. Our work refines these approaches by applying them to existing systems. Additionally, Whitaker et al [8] speculate as to the potential uses of a virtualized approach to system composition, drawing strongly on early microkernel efforts in Mach [9] among others [10, 11]. 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 [12], allow separate OSs to share devices. While we have previously demonstrated [13] 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 [14] project, whose goals are similar to our own, have used hardware for system diagnostics [15], but defer to ‘standard mechanisms’ for isolation. Intel’s SoftSDV [16], 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 [17] 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 [18] 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 and several OSs.

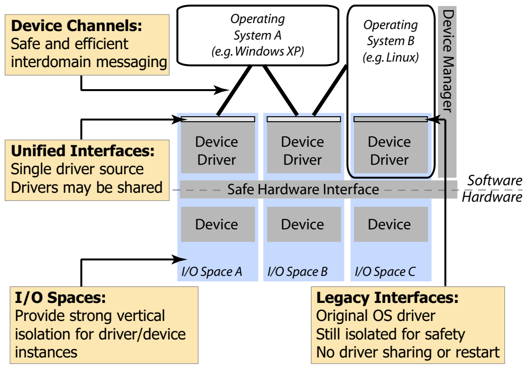

Figure 2: Example of the next-generation I/O architecture.

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 [19] and L4 [20] have investigated wrapping Linux device drivers in customized interfaces [21, 22]. Although the structure of our architecture is not entirely dissimilar to that of a microkernel, our intent is to solve the driver interface issue for all operating systems on the PC architecture, rather than make some set of existing drivers work for a single developmental OS.

The LinuxBIOS project [23] replaces the proprietary BIOS on systems with a specialized Linux image. This approach allows fast startup and eases management, especially in cluster environments where console access is not available. This is closely related to our device control interface which we intend should be directly integrated with the BIOS.

3 Architecture

In this section we outline a new architecture that addresses the dependability, maintainability and manageability of devices and their control software. Note that we incorporate hardware modifications where they are necessary or desirable, deferring until Section 4 a more pragmatic design that allows for the limitations of existing systems.

As illustrated by Figure 2, our architecture comprises three parts which correspond directly to the problems identified.

***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

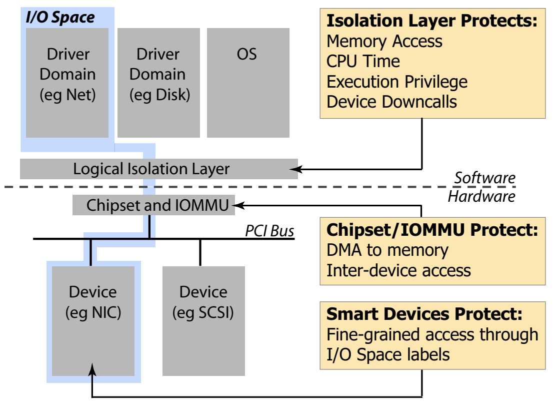

Figure 3: Achieving Safe Hardware. The outlined region denotes an *I/O Space*: a vertical slice through the system providing isolation for device and driver.

Firstly, we introduce *I/O Spaces* which arrange that devices perform their work in *isolation* from the rest of the 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 behaviour.

Previous attempts at driver isolation [2] 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. 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 becomes livelocked, or attempts to access a memory location outside its heap, then this is disallowed by the VMM and signalled to the device manager (described in Section 3.3) which takes appropriate remedial action.

Importantly, our approach also addresses the problem of isolating physical device access. Currently, drivers may write to arbitrary device registers or mask inappropriate interrupts, and devices may DMA to invalid memory. We introduce *I/O Spaces*, which extend the notion of logical protection domains to incorporate resources specific to devices and their driver code (Figure 3).

An I/O Space is a vertical slice through the system, providing an isolated context for each device and its driver. A controller within the I/O chipset maintains tables of access permissions for each I/O Space, identifying accessible ranges of memory addresses, device registers and interrupt lines. Each I/O Space is represented by a numeric identifier that is attached to every I/O transaction by a device or driver operating within that Space. The controller uses this identifier to validate the requested operation against the appropriate permission table.

Further advantages can be gained by allocating a range of I/O Spaces to each device. Incorporating a notion of *client identity* into each I/O Space would enable features such as client scheduling within the device, and safe DMA to application buffers [24]. It is also a small step to predicate bus arbitration on the requesting I/O Spaces, thus allowing differentiated service to be provided to different clients.

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 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 [15, 16, 17] 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 [18]. 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 [25].

Our unified device interfaces are based on those provided by the Xen VMM which, as we have previously demonstrated [13], 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 4.2. However, we were 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. [26]) have independently ported new devices to our architecture with minimal effort.

The end-to-end argument [27] has been invoked by efforts in the past, particularly Exokernels [7], as motivation to expose rather than to abstract hardware interfaces. 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 organisations 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 2 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 [23].

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 avail-

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

ability to OSs, and exporting configuration and control interfaces to either a local OS or to a remote manager.

4 Design and Implementation

We have implemented our next-generation I/O architecture for current PC hardware, based on the Xen virtual machine monitor. As described in [13], Xen divides the resources of a PC system amongst a set of secure and performance-isolated *domains*, each of which runs a separate guest operating system and applications. Xen implements only isolation *mechanisms*: management tasks, such as domain creation and resource allocation, are performed by a *system controller* running in a special domain with access to a privileged control interface.

We begin this section by describing how we extended Xen's virtual-machine and control interfaces to allow safe access to hardware. By placing device drivers in a resource-controlled domain separate from OS and application code, configured with suitably restrictive hardware-access privileges, Xen provides the isolation of processor and hardware contexts that we identified in requirements one and two of Section 3.1. We incorporate the necessary control and management services, provided by a *device manager* in our architectural outline, into a device-management subsystem of the system controller.

We then proceed to describe how guest OSs *connect* to drivers in other domains. We introduce an efficient method for inter-domain communication based on shared memory and asynchronous notifications, and outline the protocol for setting up *device channels* using a core interface that links every domain to the system controller. Device channels provide a unified abstraction for high-performance data transfer: we describe how this abstraction is safely implemented by device drivers and used by guest OSs.

4.1 Safe Hardware Interface

Our safe hardware interface enforces isolation of device drivers by restricting the hardware resources that they can access. To this end, 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 misbehaviour 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.

4.1.1 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 usually 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. Since the bitmap is large and sparse, for each domain Xen tracks and rewrites only the active words within the bitmap.

4.1.2 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, the domain is able to prevent unnecessary upcalls for partially-handled event sources. When the mask is eventually cleared the domain can reread the pending flag to see whether another batch of work has arrived from the event source.

Each domain specifies a single upcall handler for all event-channel notifications. To avoid the expense of linearly scanning all pending flags, a *selector word* indicates which aligned groups of 32 channels are pending. This two-level hierarchy permits fast scanning in the common situation that few channels are pending.

To avoid unbounded reentrancy, a level-triggered interrupt line must be masked at the interrupt controller until all relevant devices have been serviced. Because of this, 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. This obviates the need for unmask downcalls, as they are not required for edge-triggered interrupt lines.

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 [28].

4.1.3 Device-to-Host Interactions

As well as preventing a device driver from circumventing its isolated environment, we must also protect against possible misbehaviour 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. 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.

It is entirely reasonable to expect that IOMMU functionality will be included in commodity chipsets in the near future: 64-bit systems already include an IOMMU that allows 32-bit devices to access the full range of host memory. Although the IOMMU is intended to perform only translation, and devices are not prevented from bypassing its memory window, it is a small step to require that all memory transactions pass through the IOMMU and thus also use it to enforce protection.

4.1.4 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 only read and write registers belonging 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.

The system controller is permitted access to the entire con-

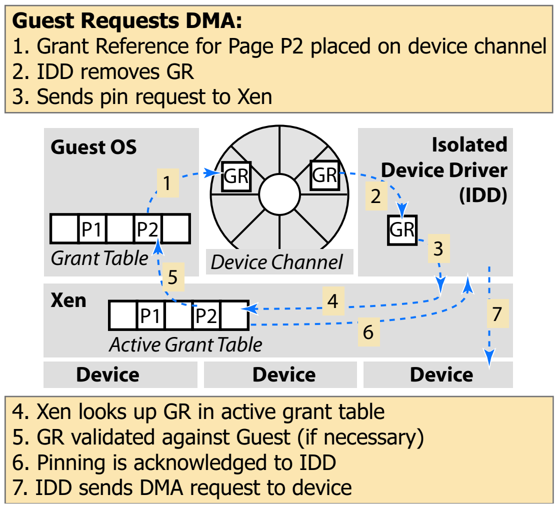
Guest Requests DMA:
1. Grant Reference for Page P2 placed on device channel
2. IDD removes GR
3. Sends pin request to Xen
Guest OS
Isolated Device Driver (IDD)
GR
GR
GR
P1
P2
Grant Table
Device Channel
Xen
P1
P2
Active Grant Table
Device
Device
Device
1
2
3
4
5
6
7
4. Xen looks up GR in active grant table
5. GR validated against Guest (if necessary)
6. Pinning is acknowledged to IDD
7. IDD sends DMA request to device

Figure 4: Using device channel to request a data transfer.

figuration space. This eases configuration of the safe hardware interface because the configuration space comprehensively describes the hardware resources that belong to each device. Thus the controller can automatically configure the correct access permissions for a new device-driver domain without assistance from the system administrator.

4.2 Device Channels

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

Guest OSs access a device through a *device channel* link with its IDD. The channel is a point-to-point communication link through which each end can send messages asynchronously to the other. A device channel is established by using the system controller to introduce the IDD to the guest OS, and vice versa. To make this possible, the system controller automatically establishes an initial control channel with each domain that it creates. Figure 4 shows a guest OS requesting a data transfer through a device channel. The individual steps involved in the request are discussed later in this section.

Instead of treating IPC as a fundamental primitive, as in most compartmentalized systems [8, 20], 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.

4.2.1 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 *grant table*. Every domain owns a private grant table that it shares only with Xen, in which each entry is a tuple $(grant, D, P, R, U)$ permitting 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 private table that is 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 writeable 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 uses of grant references by associating a usage count with each AGT entry. Whenever a foreign mapping is created with reference to an existing AGT entry, Xen increments that entry's 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 system controller.

The system controller 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 system controller. At this point the system controller may choose to destroy and restart the driver, thereby forcibly reclaiming the foreign mappings.

4.2.2 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. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows the IDD to unambiguously reorder I/O operations due to scheduling or priority considerations.

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 4.1.2. 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 the other, or mask incoming notifications.

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.

4.2.3 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, which includes fast network interfaces and disc arrays, 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 its hardware device, the IDD must first *pin* the data buffer. As described in Section 4.2.1, we enforce driver isolation by requiring the guest OS to pass a grant reference in lieu of the buffer address when requesting a device transfer: 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 4, 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 4.1.3), pinning would trigger allocation of an entry in the IOMMU. This is the only modification required to enforce safe DMA (Requirement 3 in Table 1). Moreover, this modification affects only Xen: the IDDs are unaware of the presence or otherwise of an IOMMU (in either case pin requests return a bus address through which the device can directly access the guest buffer).

In addition to pinning a guest buffer for DMA, an IDD may also use the grant reference to map the buffer into its address space. This allows support of legacy devices which do not support DMA; the IDD instead uses the main CPU to transfer data to or from the device. Buffer mapping is also useful for network-driver domains, which may choose to copy each packet header into a guest-inaccessible buffer before applying filtering rules. In this case the driver will usually both map and pin the packet buffer: all modern network interfaces support scatter-gather DMA, so they can transfer the packet payload directly from the guest buffer.

4.2.4 Device Sharing

Since Xen can simultaneously host many guest OSs it is essential to consider issues arising from device sharing. The control mechanisms for creating and destroying device channels naturally support multiple channels to the same IDD. In this section we describe 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. The low-level scheduling provided by the elevator, and also by many disc controllers, gives us good throughput, while request batching provides 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.

Sharing a high-performance network-receive path requires careful design because, apart from a few smart network interfaces that perform packet demultiplexing in hardware [24], it is not possible to DMA directly into a guest-supplied buffer. Rather than 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 overheads 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 device channel. If no unused pages are queued on the guest's network-receive ring then the packet is dropped.

For safety, Xen does not permit IDDs to exchange ownership of arbitrary memory pages. Instead we extend the grant table (Section 4.2.1) to include *exchange tuples*, $(\textit{exchange}, D, P, Q)$, permitting domain D to acquire ownership of page P in exchange for relinquishing ownership of another page. Unused exchange tuples are denoted by Q=0; when the table entry is used, Xen rewrites Q with the address of the page that was relinquished by the IDD. When exchanging page ownerships, the IDD sends a request $\textit{exchange}(P, G)$ to Xen, where P is a page frame belonging to the IDD, and G is an *exchange grant* passed to the IDD by the guest OS in a network-receive descriptor.

4.3 Control and Management

We embed our device manager within 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 device manager's ongoing responsibilities then include per-guest device configuration, managing setup of device channels, providing interfaces for hardware configuration, and reacting to driver failure.

4.3.1 Guest Configuration and Bootstrap

The device manager extends the domain-management functions of the system controller by allowing configuration of restricted IDD access for each guest OS. For example, a network-device channel may be prevented from sending packets with a spoofed source address, or a block-device channel may be limited to isolated regions of a shared disc.

As each guest OS boots, the manager informs it of the devices to which it has been granted access. The guest OS then initiates device-channel creation by allocating a memory page for the I/O-communications ring and passing a grant reference to the IDD via the device manager. The manager allocates an inter-domain event channel linking the guest OS to the IDD, and passes one endpoint of this channel, together with the grant reference, to the IDD. When the IDD acknowledges setup of the device channel, the response is forwarded to the guest OS together with its event-channel endpoint.

4.3.2 Driver Failure and Restartability

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

The recovery phase comprises several stages. First, the device manager destroys the offending IDD and replaces it with a freshly-initialized instance. The manager then signals to the connected guest OSs that the IDD has restarted; each guest is then responsible for connecting itself to the new device channel, using the normal signalling mechanisms provided by the device manager. At this point, the guest may also opt to reissue requests that may have been affected by the failure (i.e., outstanding requests for which no response was received before the IDD failed).

We have implemented OS-specific network and block-device drivers that are able to recover from an IDD restart. During recovery, the drivers retain a small amount of state to reestablish channel connections and reissue incomplete requests. State held within the shared device-channel memory (which may have been corrupted by the failed IDD) is discarded, increasing our tolerance to potential driver faults.

5 Evaluation

In this section we present an evaluation of our prototype implementation. We begin by evaluating the impact of our isolation mechanisms on realistic application workloads using industry standard benchmarks such as Postmark [29], SPEC WEB99[2] and OSDB[3]. We then proceed to investigate our overheads on individual device subsystems using

[2]http://www.spec.org/osg/web99/

[3]http://osdb.sourceforge.net/

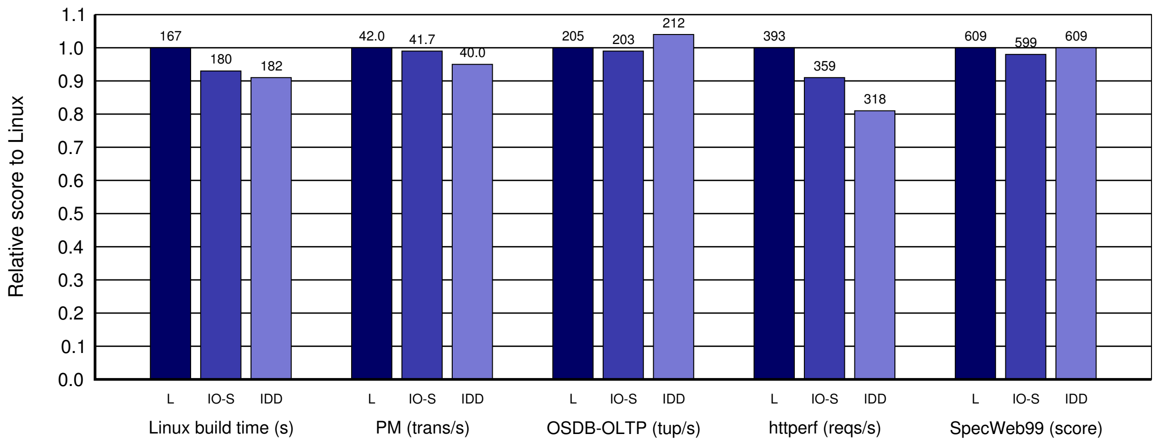

Figure 5: Application-Level Benchmarks. (L=L-SMP, IO-S=IO-Space, IDD=IDD-SMP)

a series of network and disk micro-benchmarks to determine the overhead of implementing the I/O Space protected hardware interface, and the additional overhead of device-driver isolation via the device channel interface. Finally, we provoke a series of device-driver failures and measure system availability while recovering.

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 throughout our experiments, allowing us to measure only those performance variations that are caused by varying the I/O system configuration.

Our current implementation of Isolated Driver Domains (IDDs) is based on the Xen virtual machine monitor. Xen's paravirtualized interface results in a very low virtualization overhead for CPU and memory intensive applications. When running benchmarks like SPEC CPU2000 over Xen, the measured overhead is less than 1% [13]. In the following experiments we can therefore attribute any slow down relative to native Linux to our I/O modifications. We believe this establishes a good upper bound on the additional costs, but note that the overhead may reduce if future x86 processors provide hardware assistance to reduce the cost of virtualization, or if client-aware devices reduce the load on the host processor.

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, which provides virtualization of interrupts and segregation of hardware access. We label these results `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. Each IDD and Xen/Linux instance runs in its own isolated Xen domain; the CPU to which each domain is bound depends on the test configuration. We include results in which the Xen/Linux instance executes on the same CPU as the IDDs (`IDD-UP`), in which the Xen/Linux instance runs on a different physical CPU (`IDD-SMP`), and in which the Xen/Linux instance is bound to a different 'hyper thread' within the same physical CPU (`IDD-HT`).

5.1 Application-Level Benchmarks

We subjected our test systems to a battery of application-level benchmarks, the results of which are displayed in Figure 5. 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. Postmark reports three figures for each configuration: the number of transactions per second, and the aggregate read and write throughputs. Since the relative results for all three metrics are very similar, we present only transaction rates. The additional overhead incurred by I/O Spaces and the full IDD architecture are just 1% and 5% respectively.

OSDB 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, in multi-user mode, exercises two types of workload: Information Retrieval (IR), which performs read-only operations over the entire database; and On-Line Transaction Processing (OLTP), which 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 only present the results for the OLTP workload as IR does not generate significant disk activity. We investigate the surprisingly high result achieved by IDD in our disk microbenchmark in Section 5.3.

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.

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.

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

For our experimental setup we used the Apache HTTP server version 1.3.27 with the *modspecweb99* plug-in to perform most of the dynamic content generation (SPEC rules require 0.5% of requests to use full CGI, forking a separate process). 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.

	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-UP	897 (0%)	843 (-5%)	436 (-46%)	379 (-53%)
IDD-HT	897 (0%)	897 (0%)	651 (-19%)	577 (-29%)
IDD-SMP	897 (0%)	898 (0%)	778 (-3%)	663 (-18%)

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

5.2 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, which is the system bottleneck (Table 2). The 552 byte MTU provides a far more demanding test, exposing the different per-packet CPU overheads between the configurations. The virtualized interrupt dispatch model provided by I/O Spaces incurs an overhead of 11% on transmit and 5% on receive. This shows that, even under extreme load, retaining safe control of interrupt dispatch and device acces can be achieved at reasonable cost.

The figures for our IDD implementation reflect the extra CPU cost of full driver isolation. The single CPU result represents close to a worst case scenario, recording performance slow downs of around 50% relative to `L-SMP`. This reflects the cost of rapid switching between protection domains and the deleterious effect that this has on the cache and TLB. Enabling hyper-threading (`IDD-HT`) on our single-CPU configuration provides some relief, avoiding the context switching and allowing a better data-flow through the processor's cache. Adding a second CPU (`IDD-SMP`) provides further benefits, reducing the relative deficit to 3% on transmit and 18% on receive.

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

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

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

5.4 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 6 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. Closer examination revealed that much of this period is spent executing the device driver's media detection routines while determining the link status.

During driver restart, packets that are received at the network interface are lost. A 275ms average recovery time can interact with TCP's congestion control mechanism to cause a longer effective interruption to service. We have performed experiments running full-rate TCP connections between two hosts while restarting the network driver. In this scenario we observed the TCP connection begin RTO-triggered retransmissions during the outage, effectively adding additional time to complete system recovery.

We have repeated these experiments using a 'hot standby' driver domain that resets the network interface and readvertises device channels when signalled by the system controller. This approximately halves the network outage.

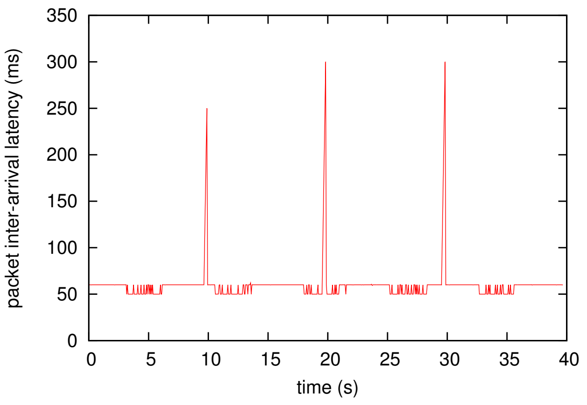

Figure 6: Effect of driver restart on packet arrivals.

Since each driver domain requires just 3MB of memory, this solution may be attractive in some scenarios.

We have also conducted similar experiments restarting block device drivers. The driver downtime in such experiments is largely determined by the time to scan the SCSI bus for devices. In situations where this is known in advance the driver could be restarted with a specific list of devices, avoiding the scan time.

6 Conclusion

We have presented a next-generation I/O architecture which solves existing problems of dependability, maintainability and manageability. Key to achieving this is the separation of device drivers from operating systems; by running each device driver in a separately protected and scheduled environment, we increase the robustness of systems to bugs in hardware and software. By using unified device driver interfaces we can *share* devices between a number of co-existing operating system instances, and dynamically *restart* device drivers in case of error or upgrade.

Unified interfaces also increase portability, allowing different kinds of operating systems to use the same device-specific code; the operating system need provide only a simple generic driver for an entire class of devices. Even when generic interfaces are inappropriate (e.g. for non-mainstream or non-shareable devices), the isolation we offer can still increase reliability and aid management.

Although the hardware required to fully support our I/O architecture is not yet available, we have implemented a prototype which makes use of a virtual machine monitor to provide the requisite functionality. The prototype supports nearly all the features of our architecture (a notable exception being protection against erroneous DMA), and gives 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] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, October 2001.

[2] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, October 2003.

[3] Microsoft calls for 64-bit driver support. InfoWorld Magazine Online, May 2004.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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).

[10] 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.

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

[12] 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.

[13] 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 Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, December 1995.

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

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

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

[24] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, April 2001.

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

[26] 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.

[27] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

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

[29] J. Katcher. PostMark: A new file system benchmark. Technical Report 3022, Network Appliance, October 1997.