Performance Optimizations for Isolated Driver Domains.

Sushrut Shirole

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science & Applications

Dr. Godmar Back, Chair

Dr. Keith Bisset

Dr. Kirk Cameron

Apr 15, 2014

Blacksburg, Virginia

Performance Optimizations for Isolated Driver Domains.

Sushrut Shirole

(ABSTRACT)

In most of today's operating system architectures, device drivers are tightly coupled with other kernel components. In such systems, a fault caused by a malicious or faulty device driver often leads to complete system failure, thereby reducing the overall reliability of the system. Even though a majority of the operating systems provide protection mechanisms at the user level, they do not provide the same level of protection for kernel components. Using virtualization, device drivers can be executed in separate, isolated virtual machines, called driver domains. Such domains provide the same level of isolation to device drivers as operating systems provide to user level applications [**?**]. Domain-based isolation has the advantage that it is compatible with existing drivers and transparent to the kernel.

However, domain-based isolation incurs significant performance overhead due to the necessary interdomain communication. This thesis investigates techniques for reducing this overhead. The key idea is to replace the interrupt-based notification between domains with a spinning-based approach, thus trading CPU capacity for increased throughput.

We implemented a prototype, called the Isolated Device Driver system (IDDR), which includes front-end and back-end drivers and a communication module. We evaluated the im-

pact of our optimizations for a variety of block devices. Our results show that our solution matches or outperforms Xen's isolated driver domain in most scenarios we considered.

Acknowledgments

Firstly, I would like to thank my advisor, Dr. Godmar Back. Under his guidance, I have acquired knowledge across a broad spectrum of computer science topics. I greatly appreciate the time he has dedicated toward our weekly meetings. I would like to thank Dr. Keith Bisset and Dr. Madhav Marathe for supporting me throughout my masters. I would like to thank Dr. Cameron for his role as a committee member and offering Computer Architecture course, which helped me learn the fundamentals.

Last but not least, I am grateful to my family for their exceptional love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A system is judged by the quality of the services it offers and by its ability to function reliably. Even though the reliability of operating systems has been studied for several decades, it remains a major concern today.

An analysis of the Linux kernel code conducted by Coverity in 2009 found 1000 bugs in the source code of version 2.4.1 of the Linux kernel and 950 bugs in version 2.6.9. This study also showed that 53% of these bugs are present in the device driver portions of the kernel [**?**].

In order to protect against bugs, operating systems provide protection mechanisms. These protection mechanisms protect resources such as memory and CPU. Memory protection controls memory access rights. It prevents a user process from accessing memory that has not been allocated to it. It prevents a bug within a user process from affecting other processes or the operating system [**?**, **?**]. However, kernel modules do not have the same level of

protection user-level applications have. In the Linux kernel, any portion of the kernel can access and potentially overwrite kernel data structures used by unrelated components. Such non-existent isolation between kernel components can cause a bug in a device driver to corrupt memory used by other kernel components, which in turn may lead to a system crash. Thus, a major underlying cause of unreliability in operating systems is the lack of isolation between device drivers and a Linux kernel.

1.1 Problem Statement

Virtualization based solutions to increase the reliability of a system were proposed by LeVasseur et. al. [**?**] and Fraser et. al. [**?**, **?**]. Fraser et. al. proposed isolated driver domains for the Xen hypervisor. In a virtualized environment, virtual machines are unaware of and isolated from the other virtual machines. Malfunctions in one virtual machine cannot spread to the others, thus providing strong fault isolation.

In a virtualized environment, all virtual machines run as separate guest domains in different address spaces. The virtualization based solutions mentioned above exploit the memory protection facilities of these guest domains. They improve the reliability of the system by executing device drivers in separate virtual machines from the kernel. Isolated driver domains are based on the split device driver model exploited by Xen [**?**].

Split device drivers are employed by Xen because its hypervisor does not include device drivers, instead it is relying on a dedicated, privileged guest domain to provide them. Xen's

split device driver model is shown in Figure 1.1. Xen uses a frontend driver in the domains

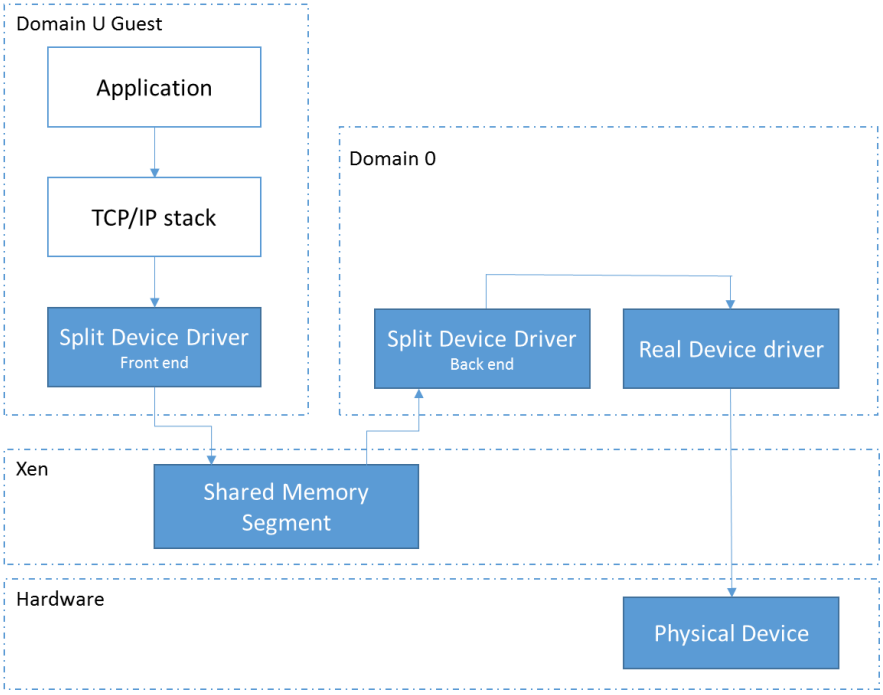


Figure 1.1: Split device driver model

that wish to access the device and a backend driver in the domain that runs the actual device driver. The frontend and the backend drivers transfer data between domains over a channel that is provided by the Xen virtual machine monitor (VMM). Within the driver domain, the backend driver is used to demultiplex incoming data to the device and to multiplex outgoing data between the device and the guest domain [**?**].

Isolated drivers use the same architecture, except that device drivers do not execute in a single, privileged domain (Domain 0), but rather a separate guest domain is created for each driver. User applications and the kernel are executed in a separate domain. As a result, device drivers are isolated from the kernel, making it impossible for the device driver to

corrupt kernel data structures in the virtual machine running user applications.

Despite advances in virtualization technology, the overhead of communication between guest and driver domains significantly affects the performance of applications [**?**, **?**, **?**]. Isolated driver domains follow an interrupt-based approach for sending notifications [**?**] between domains. The frontend and backend drivers notify each other of the receipt of a service request and corresponding responses by sending an interrupt. The Xen hypervisor needs to schedule the other domain to deliver the interrupt, which may require a context switch [**?**]. Such context switches can cause significant overhead [**?**, **?**].

1.2 Proposed Solution

In uniprocessor and multiprocessor environments, context switches may take significant amounts of time. In a multiprocessor environment, it may be more efficient for each process to keep its own CPU and spin while waiting for a resource [**?**].

In this thesis, we propose and evaluate an optimization strategy for improving the performance of the communication between guest and driver domains. We propose a solution in which a thread in the backend driver spins for some amount of time, checking for incoming service requests. Similarly, the frontend driver spins while checking for responses. On multiprocessor environments, this solution performs better than the interrupt-based approach used in the original implementation.

The source code of the isolated driver domain implementation as described by Fraser et

al [**?**] is no longer available as part of the Xen hypervisor code distribution. Therefore, we reimplemented isolated driver domains; in this thesis, we refer to our implementation as IDDR (Isolated Device Drivers). We implemented our spinning-based optimization within the IDDR framework.

We evaluated the performance of the IDDR system for multiple block device types, including ramdisks, loop devices, and hard disks attached via the Serial Advanced Technology Attachment (SATA) interface standard. Each block device is formatted with a ext2 file system and the IDDR system is evaluated by measuring the performance of the system with the SysBench benchmark. The integrity of the system is checked by executing reads and writes on the block device, with and without read ahead, as well as by performing a number of file system tests on each block device type. Our evaluation showed that the performance of the system can be improved by avoiding the context switches in the communication channel and spinning instead. IDDR trades CPU cycles for better performance. We believe this trade-off is acceptable particularly in multicore environments in which the CPU cycles used cannot easily be used by application code.

1.3 Core Contributions

The core contributions of this thesis are listed below:

1. Reimplementation of isolated driver domains, referred to as IDDR.

2. Performance optimizations for IDDR by implementing a spinning-based communication channel instead of an interrupt-based communication channel.

3. Performance comparison of the spinning-based vs. interrupt-based approaches.

1.4 Organization

This section provides the organization and roadmap of this thesis.

1. Chapter 2 provides background on processes, threads, memory protection, virtualization, hypervisors and interdomain communication.

2. Chapter 3 provides an introduction to the design of the IDDR system.

3. Chapter 4 discusses the detailed design and implementation of IDDR.

4. Chapter 5 evaluates the performance.

5. Chapter 6 reviews related work in the area of kernel fault tolerance.

6. Chapter 7 concludes the thesis and lists possible topics where this work can be extended.

Chapter 2

Background

This section provides background on operating system terminology such as processes, threads, memory protection, virtualization and hypervisors.

2.1 Processes and Threads

Process: A process can be viewed both as a program in execution and as an abstraction of a processor. Each process has its own address space [**?**].

Threads: A process has either a single or multiple threads sharing its address space. Each thread represents a separate flow of control [**?**].

A thread is also called a light-weight process. The implementation of threads and processes differs between operating systems. A process's threads share resources such as code and data

segments, whereas different processes do not share such resources.

(a) Single-threaded process (b) Multithreaded process

Figure 2.1: Single-threaded vs Multithreaded processes

2.2 Context Switches

Multiple threads typically employ time division multiplexing when sharing a single processor. In time division multiplexing, the processor switches between executing threads, interleaving their execution. The process of switching between threads is called a context switch. Continuous context switching creates the impression for the user that threads and processes are running concurrently. On multiprocessor systems, threads can also run simultaneously on multiple processors, each of which may perform time division multiplexing.

During a context switch the state of a thread is saved so that its execution can be resumed from the same point at a later time. The composition of the saved state is determined by the processor and the operating system [**?**]. The costs of context switches can be divided

into direct and indirect costs. The direct cost is the time required to save and restore processor registers, execute the scheduler code, flush TLB entries and to flush the processor pipeline. Indirect costs include subsequent cache miss penalties that are due to processor pollution [**?**, **?**].

2.3 Spinlocks

In uniprocessor and multiprocessor environments, a context switch takes a significant amount of time. In a multi-processor environment, it may be more efficient for each process to keep its own CPU and spin while waiting for a resource.

A spinlock is a locking mechanism designed to work in a multiprocessor environment. A spinlock causes a thread that is trying to acquire lock to spin if the lock is not immediately available [**?**].

Adaptive Spinning is a spinlock optimization technique. After unsuccessfully spinning for a set threshold amount of time, a thread will block as for regular locks. In the adaptive spinning technique, the spinning threshold is determined by an algorithm based on the rate of successes and failures of recent spinning attempts to acquire the lock. Adaptive spinning helps threads to avoid spinning in conditions where it would not be beneficial.

2.4 Device Drivers

A device driver is a program that provides a software interface to a particular hardware device. It enables the operating system and other programs to access its hardware functions. Device drivers are hardware dependent and operating system specific. A driver issues commands to a device in response to system calls requested by user programs. After executing these commands, the device sends data back to the driver. The driver informs the caller by invoking callback routines after receiving the data. The Linux kernel distinguishes between three device types: character devices, block devices, and network interfaces.

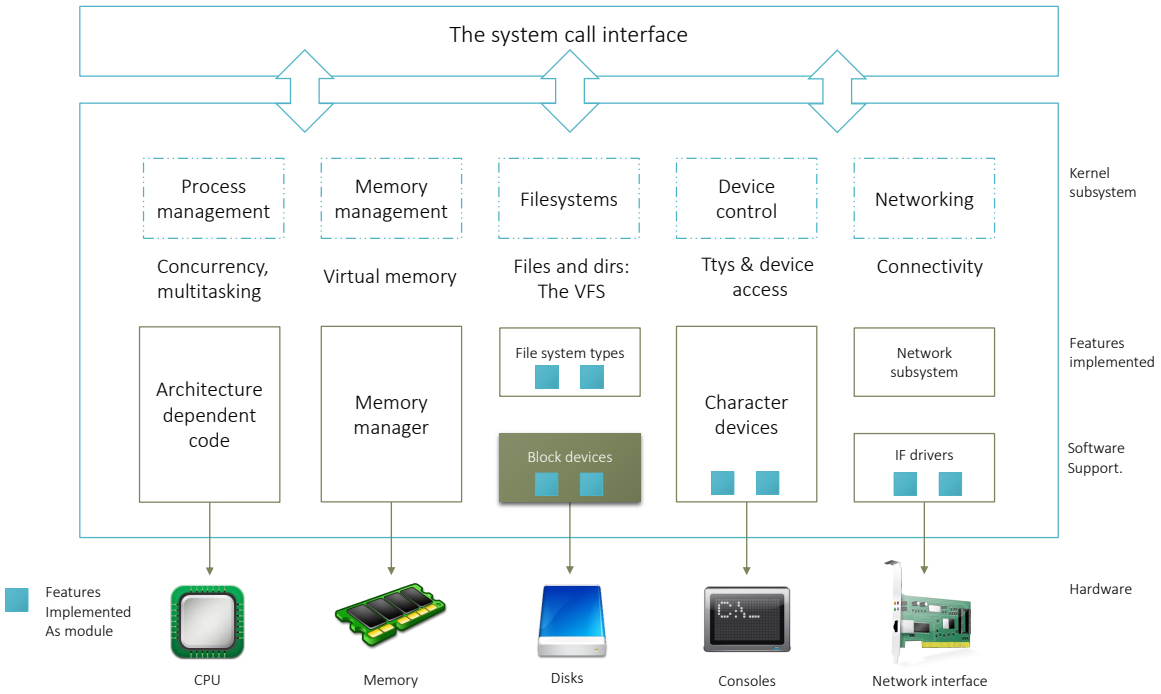


Figure 2.2: Split view of a kernel

Character devices: A character device can be accessed as a stream of bytes. A character driver usually implements at least the open, close, read, and write functions. The text console (/dev/console) and the serial ports (/dev/ttyS0) are examples of character devices.

Network Interfaces: A network interface is a device that is able to exchange data with other hosts. Usually, a network interface is a hardware device, but it can be a software device such as the loopback interface.

Block Devices: Unlike character devices, block devices are accessed as blocks of data. Whereas most Unix implementations support only I/O operations that involve entire blocks, Linux also allows applications to read and write individual bytes within a block. As a result, in Linux, block and char devices differ only in the way data is managed internally by the kernel. Examples of block devices are disks and CDs [**?**].

Block Device Drivers

Request processing in a block device driver

A block device driver maintains a request queue to store read and write requests. In order to initialize a request queue, a spinlock and a request function pointer is required. The request function forms the central part of the block device driver. Requests are added to the request queue when a request is made by higher level code in the Linux kernel, such as a file system. A block device driver's request function is called after receiving a new request. The request

function must remove all requests from the head of the request queue and send them to the block device for execution. The Linux kernel acquires a spinlock before calling the request function and releases it after returning. As a result, a request function runs in a mutually exclusive context [**?**].

A request is a linked list of `struct bio` objects. The `bio` structure contains all the information required to execute a read or write operation on a block device. The block I/O code receives the bio structure from the higher level code in the Linux kernel. The block I/O code may add the received bio structure to an existing request [**?**], if any, or it must create a new request.

Each bio structure in a request describes the low level block I/O request. If possible, the Linux kernel merges several independent requests to form one block I/O request. Usually the kernel combines multiple requests if they require access to adjacent sectors on the disk. However, it never combines read and write requests.

2.5 Memory Protection

The memory protection mechanism of a computer system controls access to resources. The goal of memory protection is to prevent malicious misuse of the system by users or programs. Memory protection also ensures that a resource is used in accordance with the system policies. In addition, it also helps to ensure that errant programs cause minimal damage [**?**, **?**].

In a Linux kernel, virtual memory is divided into pages and physical memory is divided into

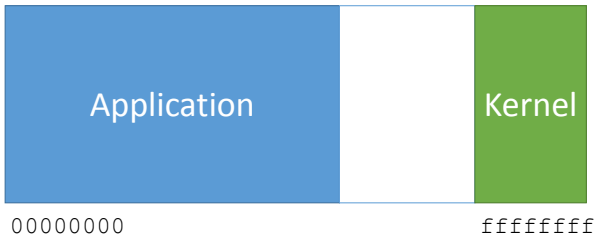


Figure 2.3: Physical memory

blocks called as page frames.

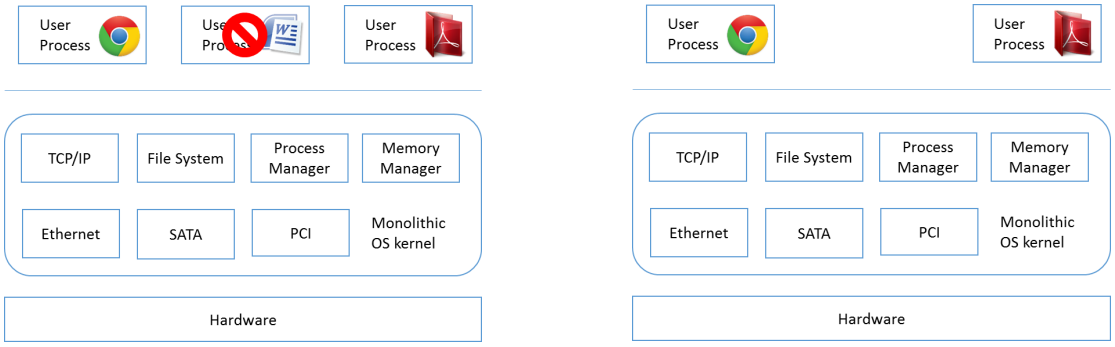
On x86-based systems, the kernel resides in the upper portion of a user application's address space. The application has access to `X Gb` of virtual address space, located at the lower end, whereas the upper `Virtual Memory size - X Gb` are reserved for the kernel. The kernel region is shared between all address spaces, allowing the kernel to access its private data structures using the same virtual address in all processes.

At the user space, applications typically run as separate processes. Each process is associated with an address space, which creates the illusion that it owns the entire memory, starting with virtual address 0 and hence the process can not access pages of other user process.

Linux kernel uses a data structure called page table to store virtual memory to physical memory mapping information. The page table entries of the upper region which is reserved for kernel are marked as protected so that pages are not accessible in user mode. However, any code running at privileged level can access the kernel memory. Hence a kernel component can access, and potentially, corrupt any kernel data structures.

Consider the example shown in Figure 2.4.

1. This system is running 3 different user processes
2. One of the processes encounters a bug and tries to access a memory address outside its address space
3. Access to the address is restricted by the memory protection mechanism

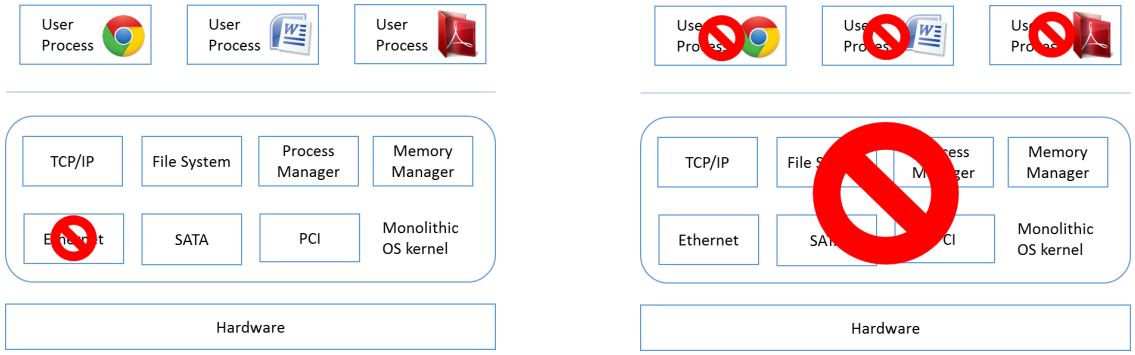


(a) A process encounters a bug (b) Other user processes are not affected

Figure 2.4: User level memory protection

Consider the example shown in Figure 2.5

1. The system runs 3 different processes in the user space and has different kernel components running in kernel space.
2. The network driver encounters a bug and corrupts kernel data structures. The corruption might lead to a system crash.



(a) A kernel component hits a bug

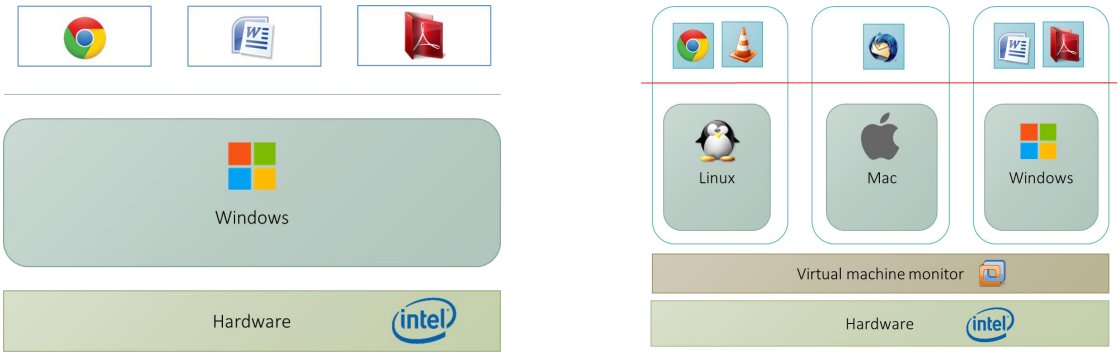
(b) Results in a system crash

Figure 2.5: Kernel level memory protection

2.6 Virtualization

Virtualization is the act of creating an abstraction of the hardware of a single machine into several different execution environments. Such abstraction creates the illusion that each separate execution environment is running its own private machine [**?**].

Virtualization provides the capability to share the underlying hardware resources and still provide an isolated environment to each operating system. Because of this isolation, any failures in an operating system can be contained. Virtualization can be implemented in many different ways, either with or without hardware support. Operating systems might require some changes in order to run in a virtualized environment [**?**]. It has been shown that virtualization can be utilized to provide better security and robustness for operating systems [**?**, **?**, **?**].



(a) Operating System Architecture (b) Virtualization

Figure 2.6: Comparison of a non-virtualized system and a virtualized system

2.6.1 Hypervisor Role

Operating system virtualization is achieved by inserting a virtual machine monitor (VMM) between the guest operating system and the underlying hardware. On the other hand, a hypervisor is a piece of computer software, firmware, or hardware that creates and runs virtual machines. Most of the literature treats VMM and hypervisors synonymously. However, whereas a VMM is a software layer specifically responsible for virtualizing a given architecture, a hypervisor is an operating system that manages VMM. This operating system may be a general purpose one, such as Linux, or it may be developed specifically for the purpose of running virtual machines [**?**].

A computer on which a hypervisor is running one or more virtual machines is defined as a host machine. Each virtual machine is called a guest machine. Among widely known hypervisors are Xen [**?**, **?**], KVM [**?**, **?**], VMware ESXi [**?**] and VirtualBox [**?**].

There are two types of hypervisors [**?**]

- Type 1 hypervisors are also called native hypervisors or bare metal hypervisors. Type 1 hypervisors run directly on the host's hardware and manage guest operating systems. Type 1 hypervisors represent the classic implementation of virtual-machine architectures such as SIMMON, and CP/CMS. Modern equivalents include Oracle VM Server for SPARC, Oracle VM Server, the Xen hypervisor [**?**], VMware ESX/ESXi [**?**] and Microsoft Hyper-V.

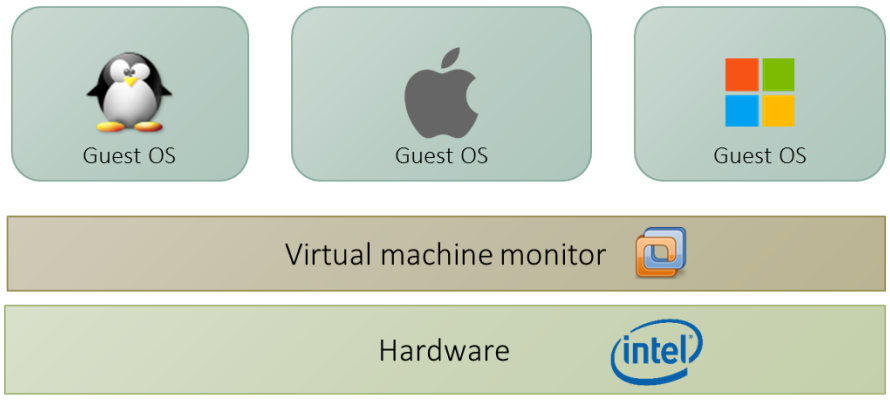


Figure 2.7: Type 1 hypervisors

- Type 2 hypervisors are also called hosted hypervisors. Type 2 hypervisors run within a conventional operating-system environment. VMware Workstation and VirtualBox are some examples of Type 2 hypervisors [**?**, **?**].

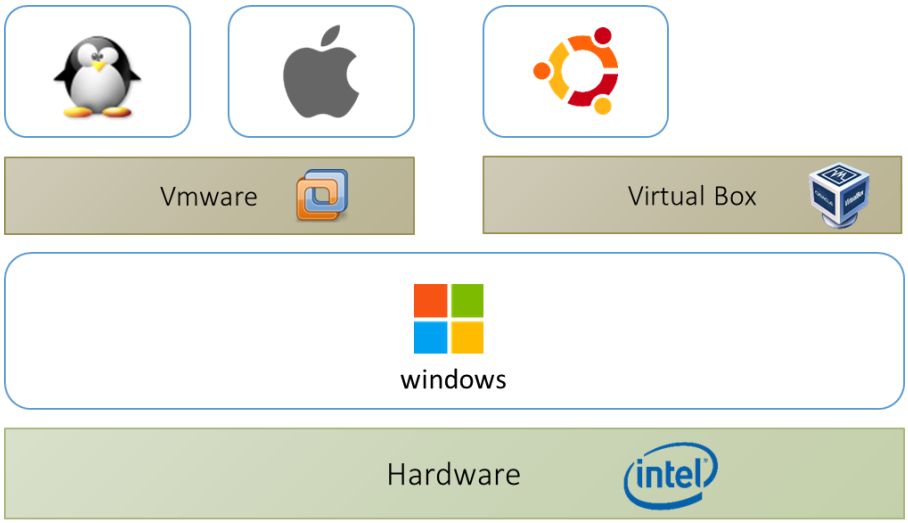


Figure 2.8: Type 2 hypervisors

2.6.2 Xen Hypervisor

Xen [**?**] is a widely known Type 1 hypervisor that allows the execution of virtual machines in guest domains [**?**], each running its own guest operating system. `Domain 0` is the first guest to run and has elevated privileges. Xen loads the `domain 0` guest kernel while booting the system. Other, unprivileged domains are called `domain U` in Xen. The Xen hypervisor does not include device drivers. Device management is included in `domain 0`, which uses the device drivers present in its guest operating system. The other domains access devices using a split device driver architecture, in which a frontend driver in a guest domain communicates with a backend driver in `domain 0`.

Figure 2.9 shows how an application running in a `domain U` guest writes data to a physical device. Xen provides an interdomain memory sharing API which is accessed through guest

kernel extensions and an interrupt-based interdomain signaling facility called event channels to implement efficient interdomain communication. Split device drivers use the memory sharing APIs to create I/O device ring buffers which exchange requests and responses across domains. They use Xen event channels to send virtual interrupts to notify the other domain of new requests or completed responses, respectively.

Consider the example shown in Figure 2.9. First, a write request is sent to the file system. After that the frontend driver puts the data to be written into memory which is shared between `domain 0` and `domain U`. The backend driver runs in `domain 0` and reads the data from the buffer and sends it to the actual device driver. The data is then written to the actual physical device [**?**].

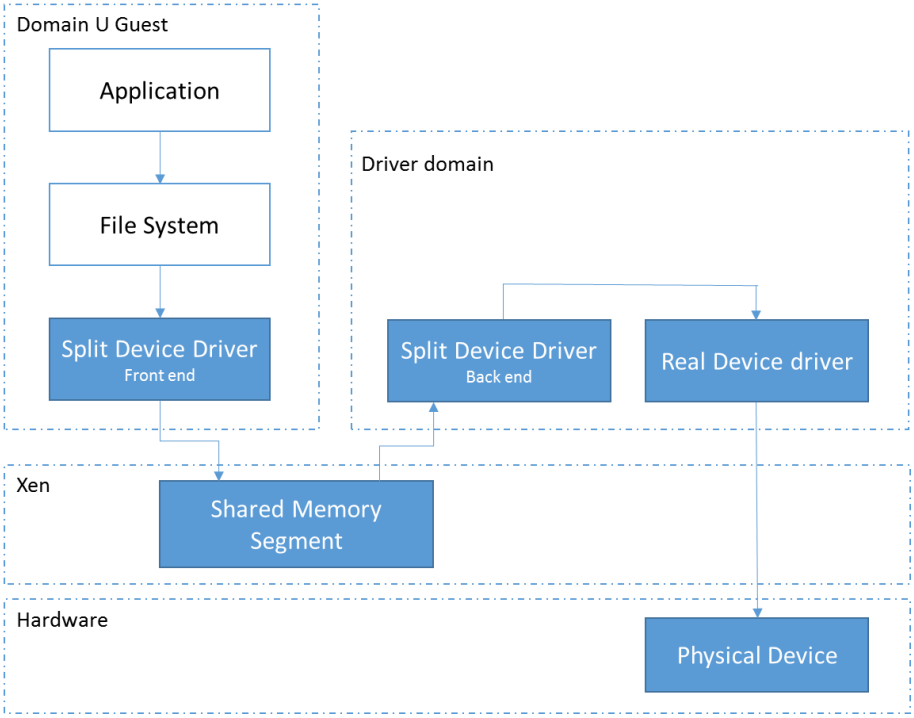


Figure 2.9: Xen split device driver

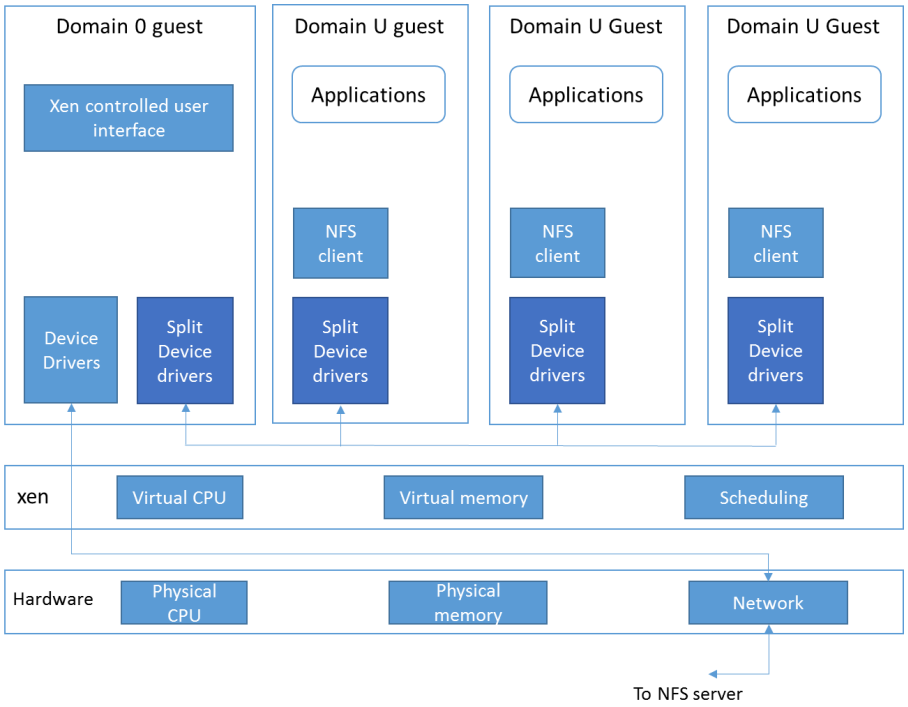


Figure 2.10: Xen

Hypercalls and Event Channels

Hypercalls and event channels are the two mechanisms that exist for interactions between the Xen hypervisor and domains. A hypercall is a software trap from a domain to the Xen hypervisor, just as a syscall is a software trap from an application to the kernel [**?**]. Domains use hypercalls to request privileged operations like updating pagetables.

An event channel is to the Xen hypervisor as a signal is to an operating system. An event channel is used for sending asynchronous notifications between domains. Event notifications are implemented by updating a bitmap. After scheduling pending events from an event queue, the event callback handler is called to take appropriate action. The callback handler

is responsible for resetting the bitmap of pending events and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen readable software flag: this is analogous to disabling interrupts on a real processor. Event notifications can be compared to traditional UNIX signals acting to flag a particular type of occurrence. For example, events are used to indicate that new data has been received over the network, or used to notify that a virtual disk request has completed.

Data Transfer: I/O Rings

Xen provides a data transfer mechanism that allows data to move between frontend and backend drivers with minimal overhead.

Figure 2.11 shows the structure of an I/O descriptor ring. An I/O descriptor ring is a circular queue of descriptors allocated by a domain. These descriptors do not contain I/O data itself. The data is kept in I/O buffers that are allocated separately by the guest OS; the I/O descriptors contain references to the data buffers. Access to an I/O ring is based on two pairs of producer-consumer pointers.

1. Request producer pointer: A producer domain places requests on a ring by advancing the request producer pointer.

2. Request consumer pointer: A consumer domain removes these requests by advancing the request consumer pointer.

3. Response producer pointer: A consumer domain places responses on a ring by advanc-

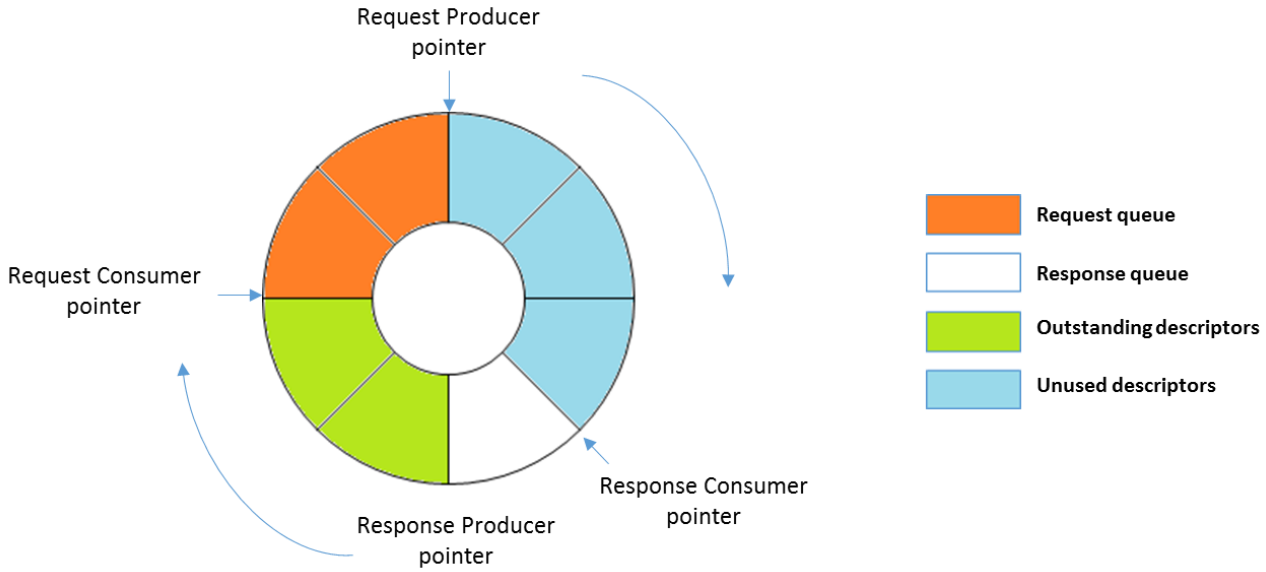


Figure 2.11: Ring I/O buffer

ing the response producer pointer.

4. Response consumer pointer: A producer domain removes these responses by advancing the response consumer pointer.

Instead of sending notifications for each individual request and response, a domain can enqueue multiple requests and responses before notifying the other domain. This allows each domain to trade latency for throughput.

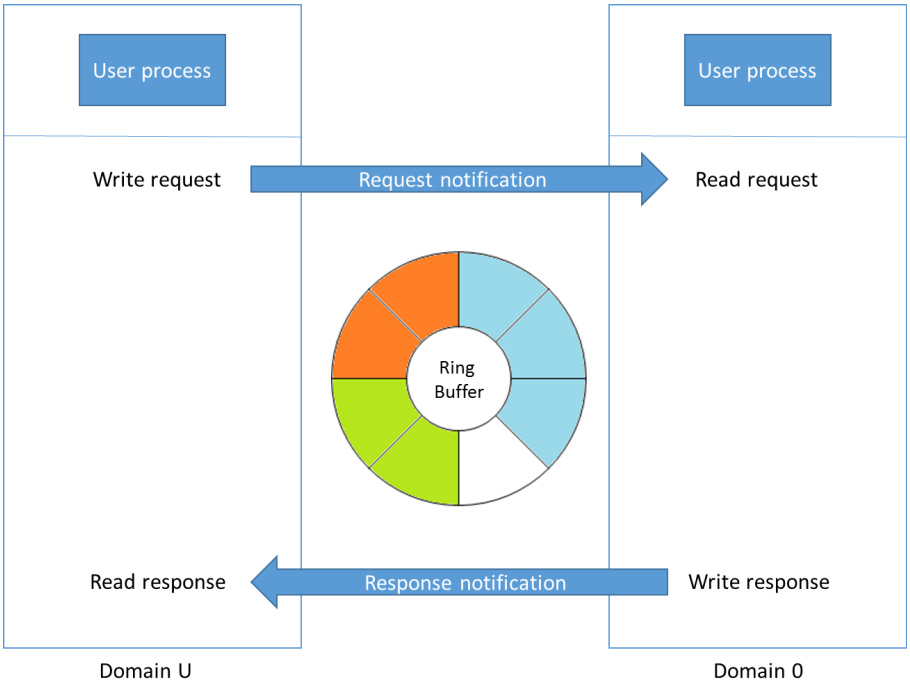


Figure 2.12: Ring I/O buffer

Shared Pages

Grant Table

Grant tables are a mechanism provided by the Xen hypervisor for sharing and transferring memory between the domains. It is an interface for granting foreign access to machine frames and sharing memory between unprivileged domains. Each domain has its own grant table data structure, which is maintained by the Xen hypervisor. The grant table data structure is used to verify the access permissions other domains have to pages allocated by a domain [**?**].

Grant References

Grant table entries are referred to as grant references. A grant reference entry contains all necessary details about a shared page. Grant references obviate the need for a domain to know the machine frame address in order to share it with other domains.[**?**, **?**, **?**]

Chapter 3

System Introduction

3.1 System Overview

Figure 3.1 presents an architectural overview of a modern operating system with a monolithic kernel and Figure 3.2 presents the architectural overview of the IDDR system.

Figure 3.2 shows that the IDDR system partitions an existing kernel into multiple independent components. User applications and the Linux kernel run in a domain called the *application domain*. A device driver, which needs to be isolated from a kernel, executes in the separate domain called the *driver domain*. Multiple domains run on the same hardware with the help of a VMM. User applications or kernel components access the hardware through the driver domain.

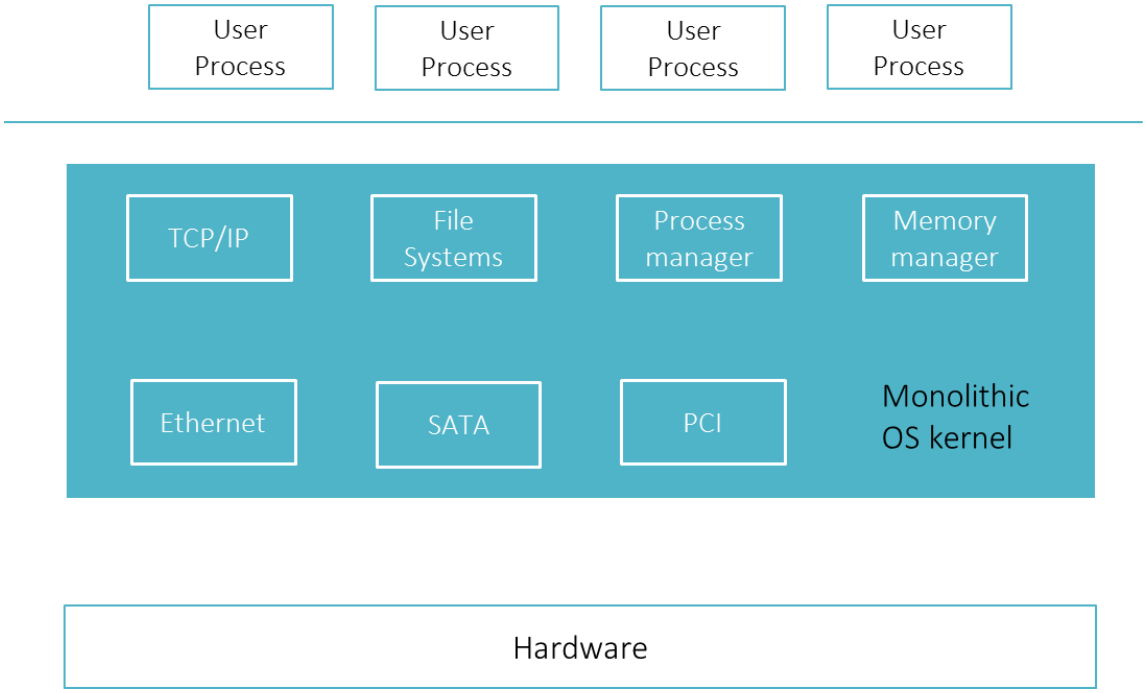


Figure 3.1: Architectural overview of a modern OS

3.2 Design Goals

The goal of isolated driver domains is to provide strong isolation between a device driver and a monolithic kernel and at the same time to avoid modifications to the device driver code. The goal of this thesis is to reduce the performance penalty due to the required communication between domains when using this approach.

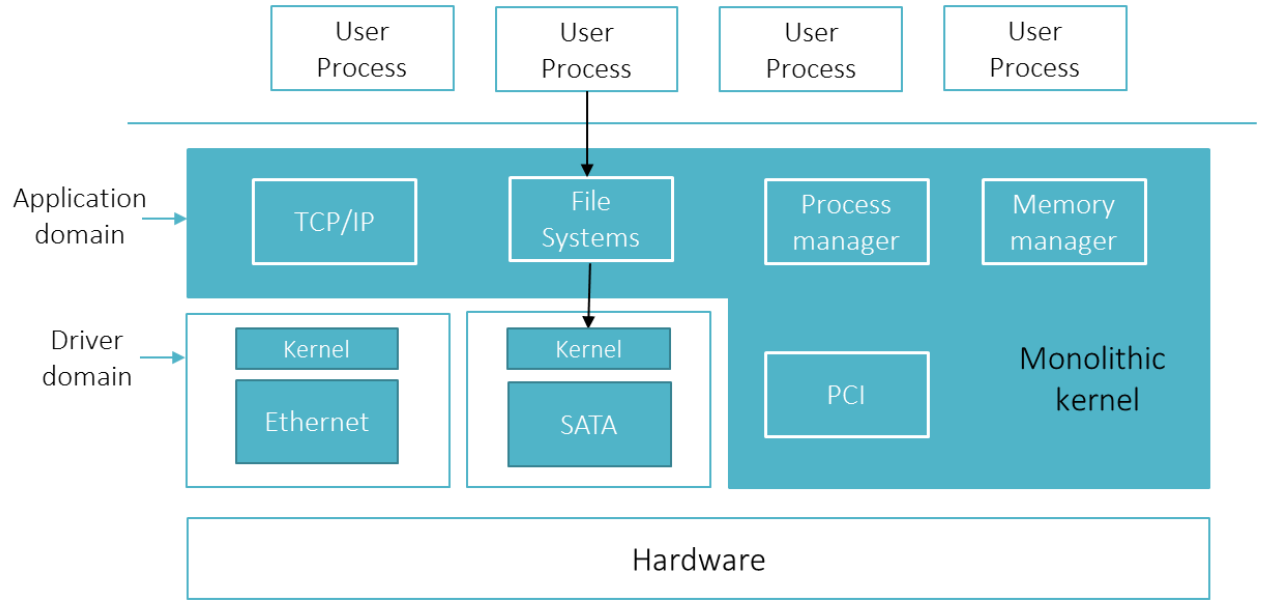


Figure 3.2: Architectural overview of the interrupt-based IDDR system

Sources of Overhead

The IDDR system is a re-implementation of isolated driver domains proposed by Fraser et. al. Even though the IDDR system provides better robustness for the operating system, it suffers from performance overheads. The main reasons for the lower performance are the data copying overhead and the interdomain communication overhead.

Copying Overhead: For write operations, the Linux kernel copies data from user space to kernel space and then from kernel space to the physical device. However, when using isolated driver domains, the system copies data first from the guest OS's user space to guest

OS's kernel space, then to a shared memory segment, and from there to the physical device. This extra copy lowers the performance of the system.

Communication Channel Overhead: The application domain and the driver domain send virtual interrupts when requests and responses are shared between both domains. These virtual interrupts cause a resumption of the receiving domain in order to deliver the them. Resuming a domain causes a context switch at the hypervisor level, which adds overhead. Our goal is to reduce this context switch-related overhead by reducing the frequency of context switches.

3.3 Design Properties

This section covers the properties of isolated driver domains, which our performance optimizations must not compromise.

Strong Isolation

Isolated driver domains provide strong fault isolation between the kernel and the device drivers. This isolation ensures that a bug within a device driver does not affect other kernel components.

Compatibility and Transparency

Alternative approaches to isolation, such as microkernels, require changes to the kernel API that is presented to applications and drivers. Consequently, existing applications and drivers will require substantial changes to work under those designs. By contrast, isolated driver domains do not change APIs visible to applications and device driver code and hence existing device drivers and applications are compatible with the new architecture. Moreover, since the frontend drivers in the application domain provides a standard device driver interface, its use is transparent to the filesystem layer using them.

3.4 System Components

This section describes the 3 main components of our design - the frontend driver, backend driver, and communication module.

3.4.1 Frontend Driver

The IDDR system runs the *frontend driver* in the application domain. The *frontend driver* acts as a substitute or proxy for the actual device driver. The main purpose of the *frontend driver* is to accept requests from user applications, process the requests, enqueue the requests for the driver domain and notify the driver domain. The *frontend driver* reads and processes the responses received from the driver domain and notifies the caller when the corresponding

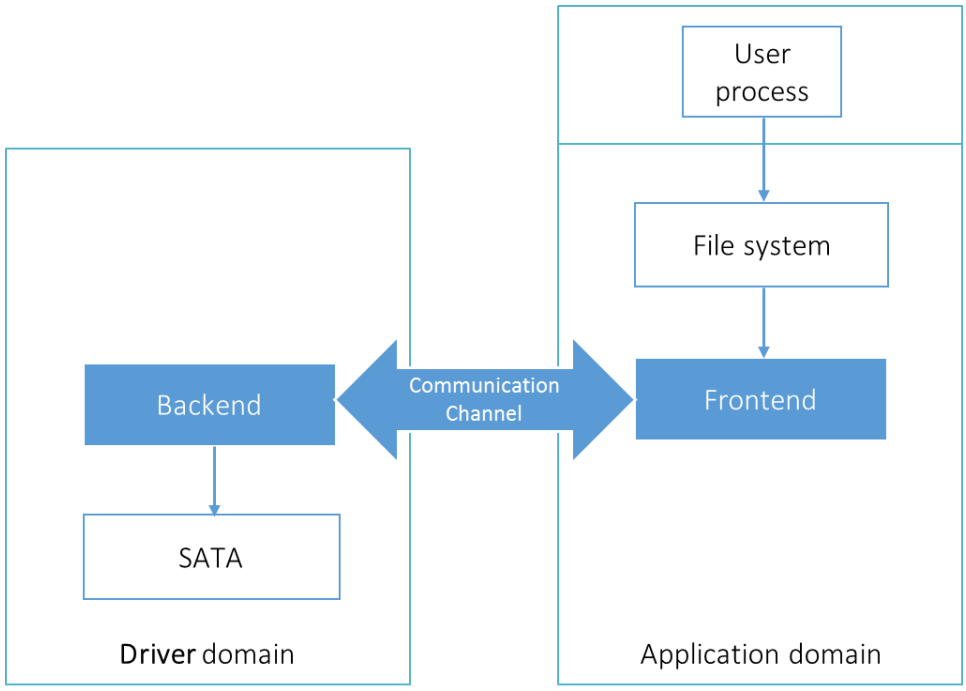


Figure 3.3: System components

requests have been completed. This notification is referred to in the Linux kernel as "ending the request".

Without the frontend device driver, we would have had to change existing applications in order to send requests to the actual device driver running in the driver domain. Hence, the frontend driver helps us achieve our transparency goal.

In the IDDR system, the frontend driver provides an interface to accept requests from a user application on behalf of the device driver. As explained in Section 2.4, each block device driver has a separate request queue to accept requests from user applications. Like all block device drivers, the frontend driver creates an individual request queue for a device to accept

requests. The frontend driver submits the requests to the communication channel. The frontend driver receives a software interrupt upon the availability of responses in the shared queue. The frontend driver handles the software interrupt by reading data from the shared memory in case of read operations and sends a completion notification to a user application. In the case of write operations, it only sends the completion notification.

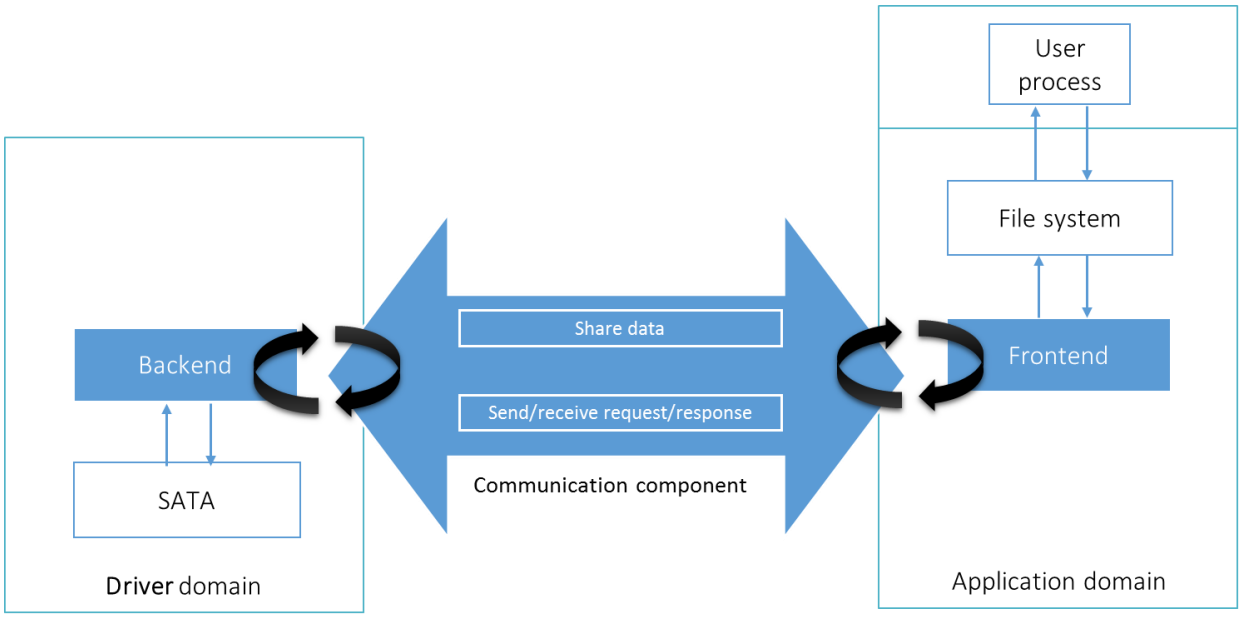


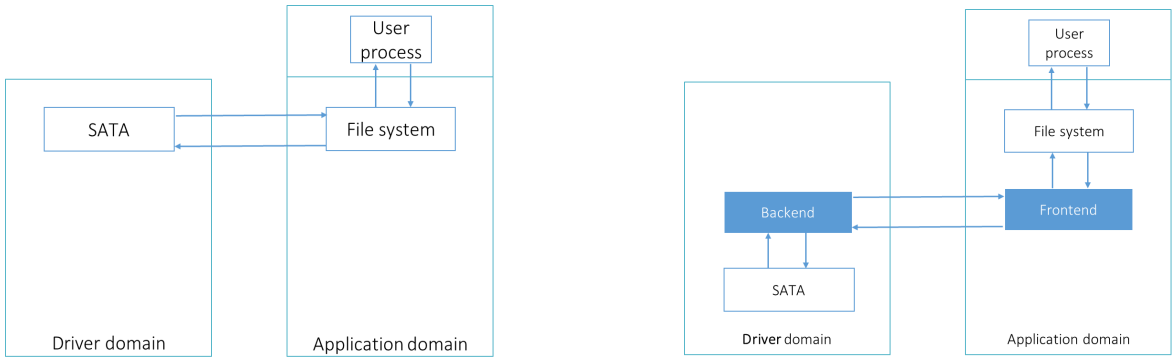
Figure 3.4: Spinning-based IDDR system

3.4.2 Backend Driver

The IDDR system runs a *backend driver* runs in the driver domain. The responsibility of the *backend driver* is to accept requests from the application domain and forward them to the actual device driver. The *backend driver* sends the responses and notifies the application

domain after receiving the responses from the device driver.

Without the backend device driver, we would have had to change existing device drivers in order to send responses to user applications running in the application domain. Hence, with introduction of the backend driver we achieve the compatibility goal.



(a) Conceptual design of the driver domain (b) Backend and frontend driver

Figure 3.5: Role of the frontend and the backend drivers

3.4.3 Communication Module

The communication module provides a communication channel between the *frontend driver* and the *backend driver*. The communication channel is logically divided into three parts. The responsibility of the communication module is to

1. share the requests and responses between the driver domain and the application domain.

2. manage and share the data of read/write requests/responses.

3. notify the domain upon the occurrence of events such as when requests or responses are produced.

Figure 3.6 provides an overview of the communication module.

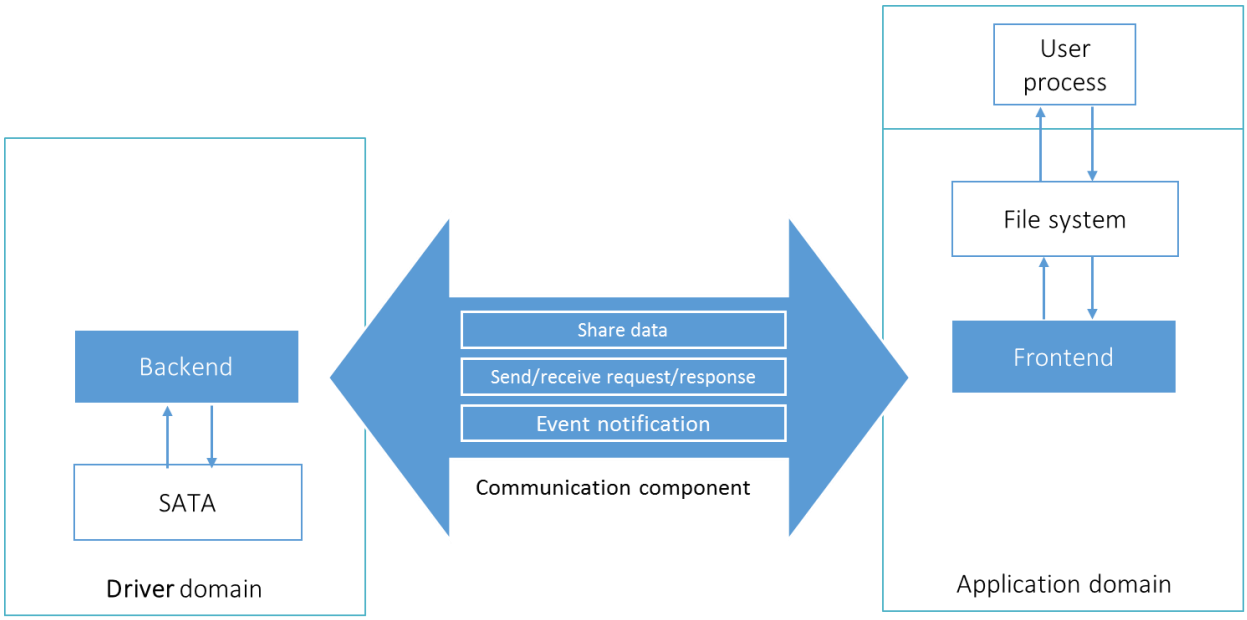


Figure 3.6: Communication module

Interrupt-based IDDR System: In the interrupt-based approach, the frontend driver submits requests to the communication module. The communication module copies the data of the write requests to shared memory. The communication module is responsible for the allocation and de-allocation of the shared memory pages. Once a sufficient number of requests is submitted by the frontend driver, the communication channel shares the requests

with the backend driver. It notifies the backend driver that requests are available in the shared request queue.

Spinning-based IDDR System: In the interrupt-based approach, by default a software interrupt is sent to the domain as a notification. Each such software interrupt causes the hypervisor to schedule the driver domain. Similarly, software interrupts which notify the availability of responses, causes the hypervisor to schedule the application domain. The scheduling of the driver domain and the application domain might result in a context switch.

In order to avoid these context switches, we run an intermediate thread in the frontend driver and an intermediate thread in the backend driver. Both threads spin for the availability of requests and responses in the shared queue. The intermediate threads delegate the responsibility of the notifications from the frontend driver and backend driver to the communication module.

Chapter 4

Implementation

This chapter describes specific implementation details of the IDDR system.

4.1 Overview

We implemented the IDDR system based on the Linux kernel, version 3.5.0 and Xen version 4.2.1. The application domain and the driver domain run the same kernel binary. Table 4.1 summarizes our implementation efforts in terms of the number of lines of code.

The IDDR system implementation did not require any changes to the device driver code. However, we did make a small number of changes to the Xen and Linux kernel.

Figure 4.1 shows the implementation overview of the spinning-based IDDR system.

Table 4.1: Implementation efforts in terms of number of lines of code.

Component	Interrupt-based IDDR	Spinning-based IDDR
Linux Kernel	6	6
Xen	252	252
Front-end Driver	611	712
Back-end Driver	692	752
Total	1561	1722

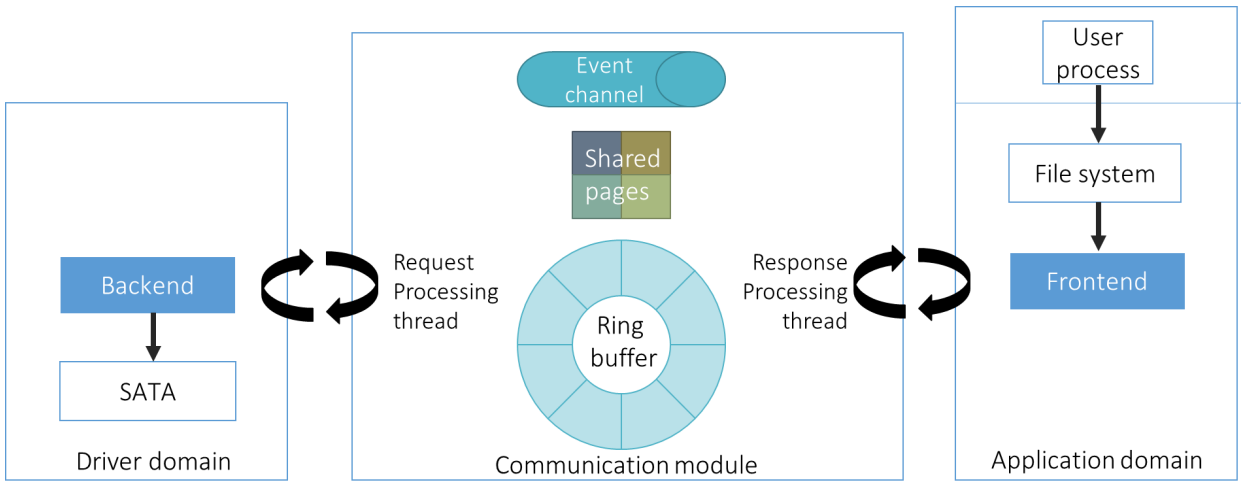


Figure 4.1: Implementation overview of the spinning-based IDDR system

4.2 Communication Module

This section describes the implementation details of the communication modules of the interrupt-based and the spinning-based IDDR systems, respectively.

4.2.1 Interrupt-based IDDR system

As Section 3.4.3 describes, the role of the communication module in the interrupt-based IDDR system is to:

1. Share requests and responses between the driver and application domains
2. Share the data associated with read/write requests/responses
3. Notify the domain upon the availability of requests and responses

Shared Request and Response Queue: In order to implement the first role of the communication module, we use the ring buffer mechanism provided by Xen. A ring buffer is a shared I/O ring, which is explained in Section 2.6.2. We divide the ring buffer into the front half and the back half. The IDDR system uses the front half as the shared request queue and the back half as the shared response queue.

The IDDR system allocates the ring buffer in the initialization stage of the communication module and initializes the ring buffer in the application domain. Whenever the frontend

driver receives a request from an application in the application domain, the frontend driver removes the request from the device driver queue and submits it to the communication module. The communication module checks if there is free space in the shared request queue, and if so, allocates the space for the new request. After batching requests together, the communication module pushes all requests to the shared request queue.

Shared Memory for Read/Write Data: We use the ring buffer only to share the control information associated with requests and responses. In order to share the actual data associated with requests and responses we use shared pages.

As explained in Section 2.6.2, a grant table is used for sharing memory between domains. We use a grant table to share memory between the application domain and the driver domain.

Event Notification: We create an event channel in the initialization stage of the communication module in the application domain and connect to it in the initialization stage of the communication module in the driver domain. We attach an interrupt handler routine for the event channel in both domains. The interrupt handler routine in the application domain reads responses from the shared response queue and forwards them to the frontend driver. The interrupt handler routine in the driver domain reads requests from the shared request queue and forwards them to the backend driver.

4.2.2 Spinning-based IDDR System

This section describes the communication module of the spinning-based IDDR system:

Read response thread in the application domain:

In the spinning-based IDDR system we create a kernel thread called the *read response thread* during the initialization stage of the application domain's communication module. The read response thread spins to check if responses are available in the shared response queue. If a response is available, it reads the response from the shared response queue. If a response is not available in the shared response queue, the thread goes into a sleep state after spinning for a set threshold duration. We maintain the status of the thread as `SLEEPING` or `RUNNING` in a data structure shared between the domains for this purpose. When the thread exceeds the spinning threshold, we change its state from `RUNNING` to `SLEEPING`, and we check again the response queue for new responses in order to avoid a race condition.

A thread must not sleep unless it is assured that it will eventually be woken up. The code doing the waking up must also be able to identify the thread to be woken up. We use Linux's standard `wait queue` mechanism to implement thread wakeup in a reliable and race condition free manner. A wait queue is a list of threads waiting for a specific event [**?**, **?**].

We initialize a wait queue for the read response thread during the initialization stage of the application domain's communication module. The read response thread sleeps in the

wait queue, waiting for a flag denoting the availability of the response to be set. The driver domain's communication module checks the status of the read response thread after pushing responses on the shared response queue. If the status is `SLEEPING` then it sends a virtual interrupt. It uses an event channel to wake the read response thread. The wakeup signal is sent in the form of an event channel notification from the driver domain to the application domain.

Similar to the interrupt-based IDDR system, we create a new event channel in the initialization stage of the communication module in the application domain. We attach an interrupt handler routine for the event channel in the application domain. In the interrupt handler, the communication module wakes up the read response thread if it is sleeping.

Read request thread in the driver domain:

The implementation of the read request thread is similar to the implementation of the read response thread, except that the read request thread spins in the driver domain waiting for requests to become available in the request queue.

4.3 Frontend Driver Implementation

Interrupt-based IDDR System

The core responsibility of the frontend driver in the interrupt-based IDDR system is to:

1. Provide an interface which appears as a block device to file systems and user applications.
2. Accept a request from a file system or a user application
3. Send the request to the driver domain
4. Send a completion notification to a user application after reading the response

The implementation of the frontend driver is split into 3 parts:

1. Initialization code
2. Submitting requests to the communication module
3. Sending completion notifications to user applications

Initialization Code: During the initialization phase, the frontend driver creates a separate interface for each block device. The interface for each block device is associated with a device driver queue. Read and write requests issued on the interface are queued in this device driver queue.

Request Submission: The frontend driver removes the request submitted to the driver interface and converts the request into a request structure, which is required by the backend driver. The request structure points to the shared memory allocated for the read/write data by the communication module. The frontend driver then forwards the newly created request to the communication module, which submits the request to the backend driver.

Completion Notifications: We maintain a shadow table that contains an entry for each request that was received in the device driver queue. We implement the shadow table as a circular array whose entries refer to these requests. We maintain an ID for each request. The backend driver copies this ID into the corresponding response. The ID is used for mapping the response to the request in the shadow table. When a response is read by the communication module, it forwards the response to the frontend driver. The frontend driver looks up the corresponding request in the shadow table, and sends a completion notification to the user application that issued the request.

Spinning-Based IDDR System

The core responsibility and implementation of the frontend driver of the spinning-based IDDR system is similar to the frontend driver of the interrupt-based IDDR system.

4.4 Backend Driver Implementation

The role of the backend driver in the IDDR system is to:

1. Read a request through the communication module and convert it to a block I/O request
2. Accept a response from the block device driver
3. Forward the response to the communication module

The implementation of the backend driver is split into 2 parts.

1. Forwarding requests to the actual block device in the required block I/O request format.

2. Forwarding responses to front-end driver.

Request Forwarding

The backend driver converts a request that is received via the communication module into the required block I/O request format (using `struct bio` objects as discussed in Section 2.4), so that the block device can process the request. In order to create a block I/O request, pages from the shared memory are mapped and inserted into the block I/O request structure and any required information is copied from the shared request into the block I/O structure. The newly created request is sent to the actual driver execution. Once the block I/O request's execution is completed, the driver calls a provided callback function.

Response Forwarding

Irrespective of the success or failure of a request's execution, the backend driver forwards the real device driver's response to the frontend driver. The result of the execution and a request ID are copied into a newly allocated response structure. The request ID is used as an index in the shadow table to map responses to requests. The communication module pushes the response into the shared response queue, so that the response is made available to the application domain.

4.4.1 Future Work

Adaptive spinning: In the current implementation the read request and read response spinning thresholds are set to be constant amounts of time. As future work, adaptive spinning could be implemented in which the threshold is varied based on assumptions about the cost of blocking and resuming the respective threads.

Chapter 5

Evaluation

We use Linux Kernel version 3.5.0 for both application domain and driver domains. We use the Arch Linux distribution for the `x86_64` platform for our testing and performance evaluation. The specifications of the system used for the evaluation are presented in Table 5.1.

5.1 Goals

The goals of our evaluation are:

1. **Comparison of Xen's isolated driver domain mechanism with the interrupt-based IDDR system:**

   In order to verify that the interrupt-based IDDR system constitutes a suitable baseline against which to compare the spinning-based IDDR system, we compared our interrupt-

Table 5.1: Hardware specifications

System Parameter	Configuration
Processor	2 X Quad-core AMD Opteron(tm) Processor 2380, 2.49 Ghz
Number of cores	4 per processor
Hyperthreading	OFF
L1 L2 cache	64K/512K per core
L3 cache	6144K
Main memory	16Gb
Storage	SATA, HDD 7200RPM

based implementation against Xen's isolated driver domain. However, since the source code of the isolated driver domain implementation is not available, we achieve this evaluation goal by comparing the performance of the interrupt-based IDDR system with Xen's regular drivers, which use the same split device driver model.

2. **Performance impact of spinning-based optimizations:**

   The second goal of the evaluation is to prove that the spinning-based implementation of the communication channel improves the performance of the interdomain communication and hence the IDDR system.

   We achieve this evaluation goal by comparing the performance of the interrupt-based IDDR system with the spinning-based IDDR system.

5.2 Methodology

In order to measure the performance of the system, we run performance tests using a variety of block devices. In order to cover a variety of devices we use block devices such as SATA disks, ramdisks and loop devices for the performance testing. A loop device is a device that provides a block device that is backed by a file. A ramdisk is a block of a memory that acts as a disk drive.

In order to conduct the performance tests, we format the block device with the ext2 file system, and run the SysBench benchmark [**?**] on it. SysBench is a multi-threaded benchmark tool for evaluating operating system performance. It includes different test modes, one of which is FileIO. The FileIO mode can be used to produce different file I/O workloads. It runs a specified number of threads to execute requests in parallel. We run the SysBench benchmark in FileIO test mode to generate 128 files with `1GB` of total data. We execute random reads, random writes, and a mix of random reads and writes on all three devices. We set the block size to `16KB`. We vary the number of SysBench threads from 1 to 32 to measure the throughput of the system under different levels of concurrent access.

5.3 Xen Split Device Driver vs Interrupt-based IDDR System

As per our first evaluation goal discussed in Section 5.1, we compare the interrupt-based IDDR system with Xen's split device driver.

Experimental Setup

Xen Split Device Driver

We create a ramdisk in domain 0. The guest domain (domain U) is configured to access domain 0's ramdisk via a split device driver. We use the same setup for the loop device and the SATA disk. We format and mount the disk into a partition in the guest domain using the ext2 file system in all cases. The SysBench benchmark is run on the mounted partitions as explained in Section 5.2.

Interrupt-based IDDR System

In the Xen hypervisor, the domain 0 always runs as a paravirtualized guest (PV), but domain U can be using both a hardware-assisted virtualization configuration (HVM) or a paravirtualized configuration. In our setup we run domain U always in HVM mode because HVM guests exhibit less system call overhead and faster memory bandwidth compared to

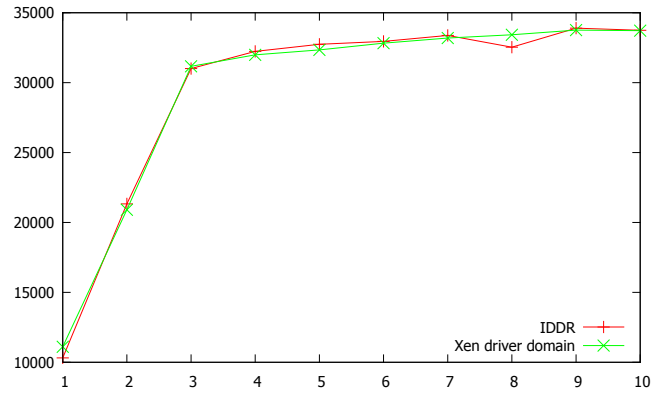
PV guests, as we observed by running the system call micro-benchmark tool lmbench[**?**].

When comparing to Xen's split device drivers, the backend device driver executes in domain 0 and the frontend device driver in domain U. To eliminate any performance differences that may be solely due the mode of virtualization used, we matched the modes of virtualization by running IDDR's backend and frontend drivers also in domain 0 and U, respectively.

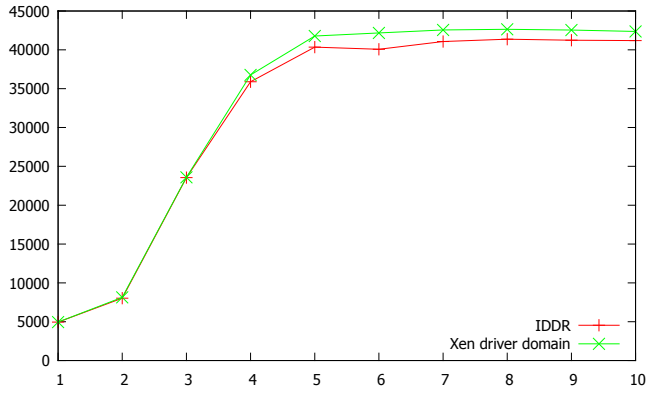
Comparison

We compare the throughput of Xen's split device driver and the interrupt-based IDDR system in Figure 5.1a and Figure 5.1b. Figure 5.1a presents the throughput of both systems when data is randomly read from a ramdisk and written to it. Figure 5.1b presents the throughput when data is randomly read from a loop device.

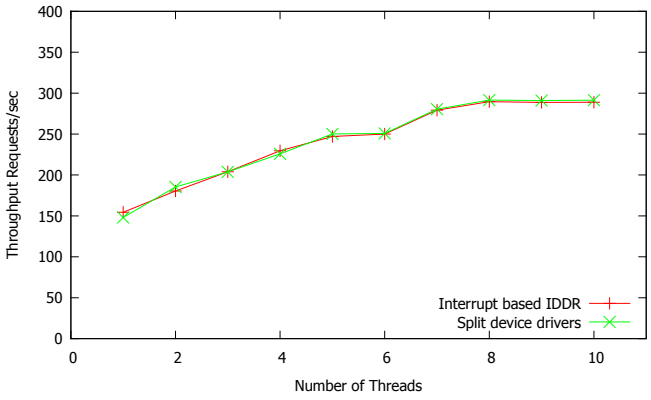
On a loop device, the performance of the interrupt-based IDDR system differs by 3%-4% when compared to Xen's split device driver. On a ramdisk, the throughput of the interrupt-based IDDR system matches that of the Xen split device driver. This shows that our interrupt-based IDDR implementation provides a suitable baseline for our performance optimizations.



(a) Random reads and writes on a ramdisk



(b) Random reads on a loop device



(c) Random reads on a SATA disk

Figure 5.1: Interrupt-based IDDR system vs Xen split driver

5.4 Interrupt-based IDDR System vs Spinning-based IDDR System

We measure and compare the performance of the interrupt-based IDDR system with the spinning-based IDDR system. To compare the performance of both systems, we measure performance of the system by varying the number of SysBench threads. The SysBench benchmark executes random reads and writes on a ramdisk, loop device, and a SATA disk.

Experimental Setup

In both systems, the application domain is domain 0, and the driver domain is domain U. We create a ramdisk and insert the backend driver in the driver domain. We insert the frontend driver in the application domain. We measure the performance of both systems on a loop device with the same setup.

However, we were unable to set up PCI passthrough for the SATA disk, preventing us from running the SATA controller driver in domain U. For the SATA portion of the experiments, we use domain 0 as the driver domain and domain U as the application domain, while still retaining all other aspects of the IDDR implementation.

Random reads and writes

Comparison: Figure 5.2, Figure 5.3, and Figure 5.4 compare the throughput of the interrupt-based IDDR and the spinning-based IDDR system for read, write, and read/write workloads using a ramdisk, loop device, and SATA disk, respectively.

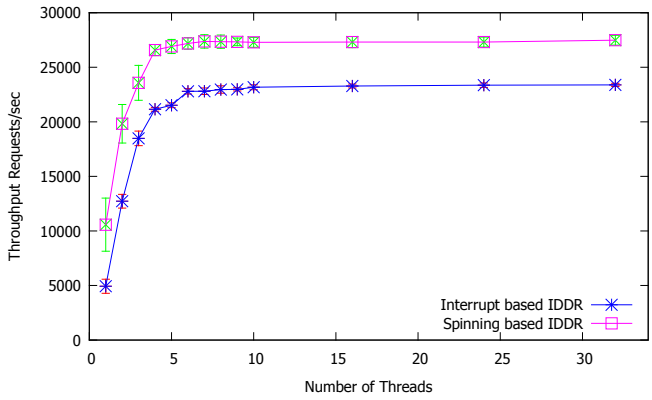
Figure 5.2a, Figure 5.3a, and Figure 5.4a show that the spinning-based IDDR system performs better when data is read from a device randomly.

Figure 5.2b, Figure 5.3b and Figure 5.4b compare the performance of the device when data is written randomly to a device. The graph shows that initially the spinning-based IDDR system performs better than the interrupt-based IDDR system, but as the number of SysBench threads increases, the throughput of the spinning-based IDDR system decreases.

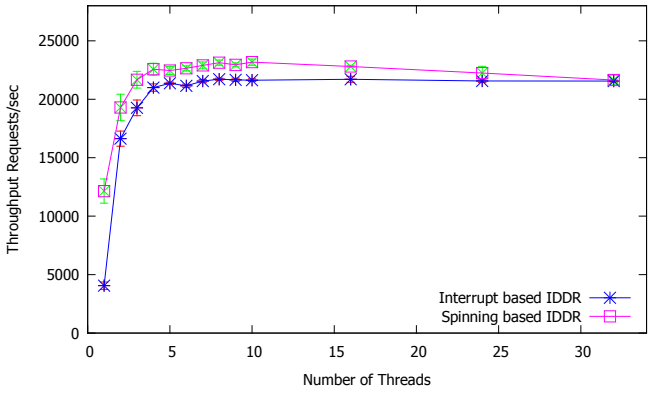
Figure 5.2c, Figure 5.3c and Figure 5.4c compare the performance for mixed random read and write workloads.

Observation: The performance analysis of the interrupt-based and spinning-based IDDR system shows that initially the throughput of both systems increases and then it remains constant. We measure the throughput of the system with varying number of SysBench threads.

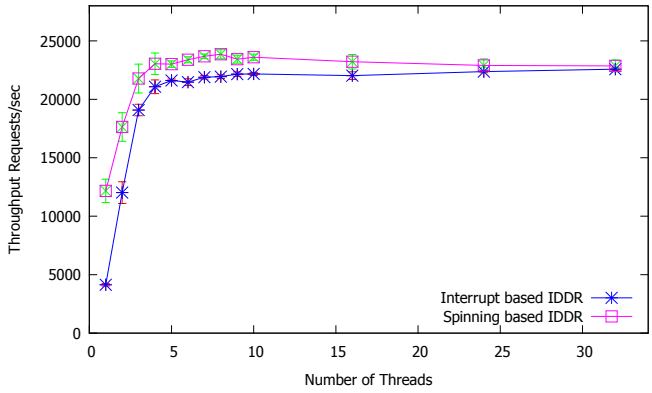
In both systems, when the number of SysBench threads is low, the rate at which data is read and written is low and when the number of SysBench threads is high, the rate at which data is read and written is high. Once maximum throughput is achieved, it remains constant.



(a) Random reads



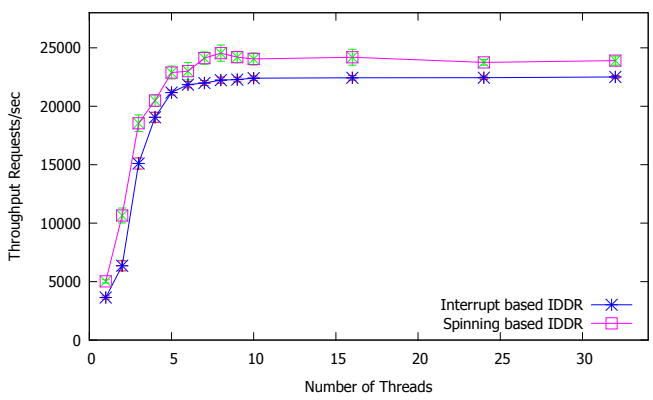
(b) Random writes



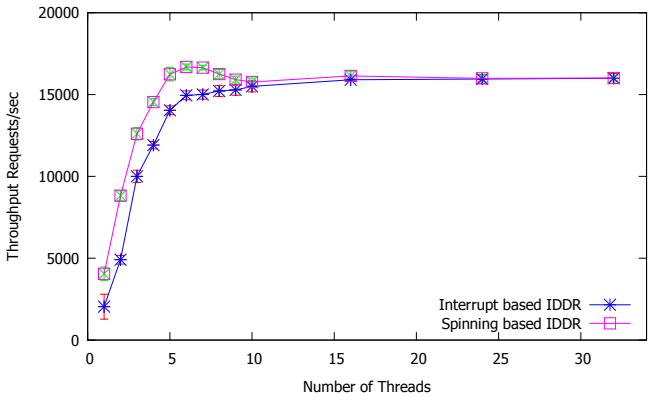
(c) Random reads writes

Figure 5.2: Random reads and writes on a Ramdisk

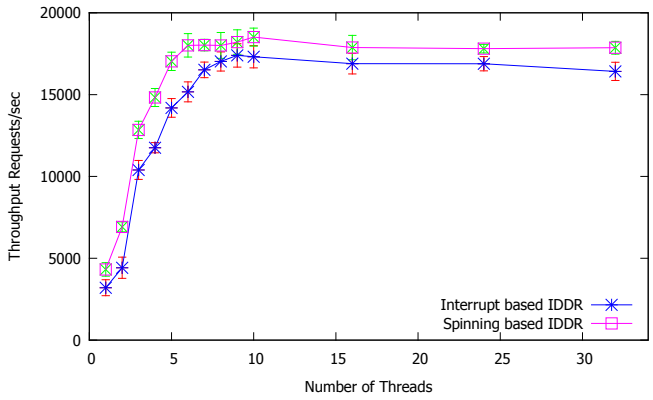
In the interrupt-based IDDR system, virtual interrupts are sent between application and driver domains whereas the spinning-based approach attempts to avoid these virtual interrupts. Our optimization technique reduces the frequency of the virtual interrupt being sent between domains. The higher performance gains at lower levels of concurrency may be due to missed opportunities for batching in the interrupt-based case, increasing the need for virtual interrupts even further.



(a) Random reads



(b) Random writes



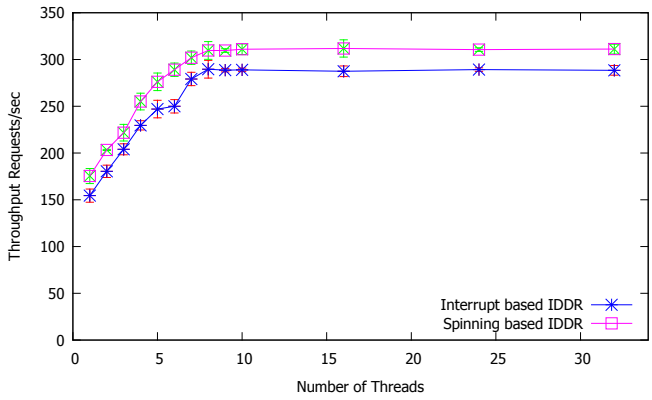
(c) Random reads writes

Figure 5.3: Random reads and writes on a Loop device

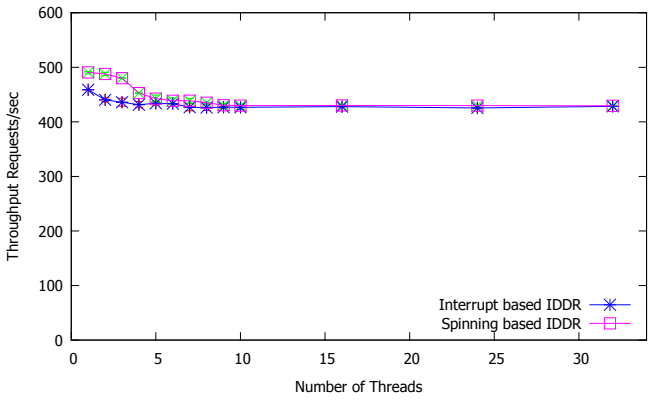
CPU utilization

The spinning-based IDDR system uses CPU cycles that could also be used by applications unless excess CPU capacity is available (as in many multicore or manycore systems). Hence we exploit a trade-off between higher CPU utilization and higher throughput.

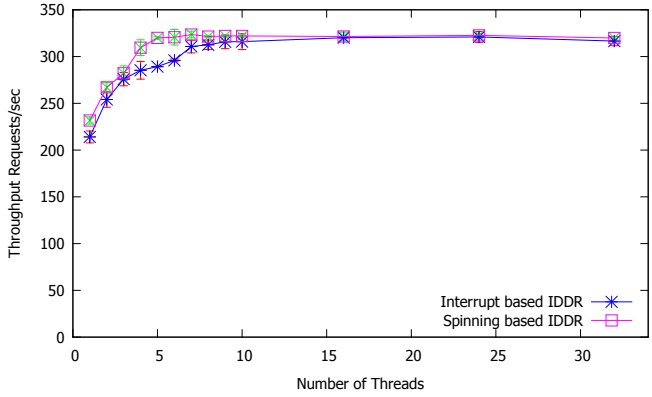
In this section we compare the CPU utilization of the interrupt-based and the spinning-based



(a) Random reads



(b) Random writes



(c) Random reads writes

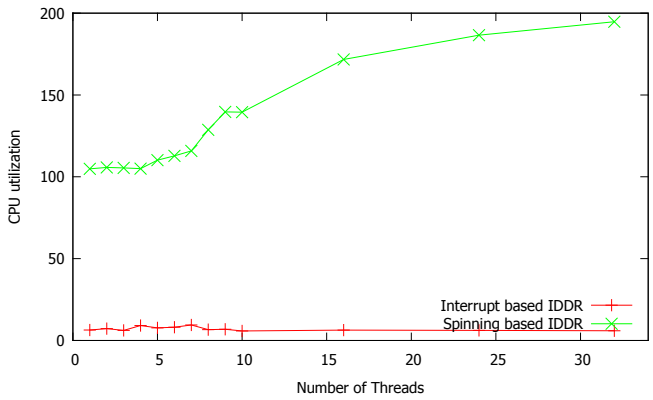
Figure 5.4: Random reads and writes on a SATA disk

IDDR systems.

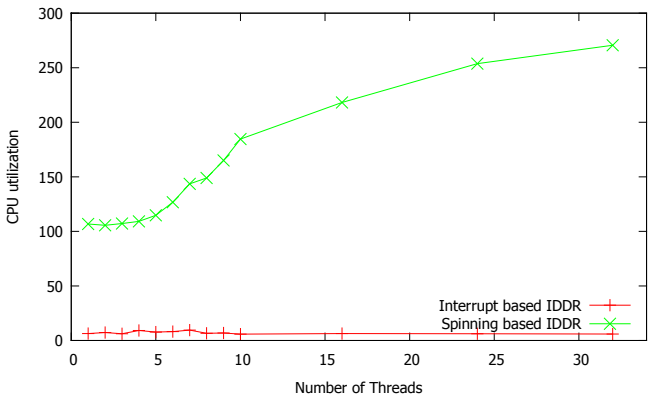
Figure 5.5, Figure 5.6, and Figure 5.7 compare the CPU utilization of both systems using the ramdisk, loop device and SATA disk setups.

Observation: In the spinning-based version of IDDR, the read response and request threads spin continuously unless the driver is idle for extended periods of time. Thus, up to

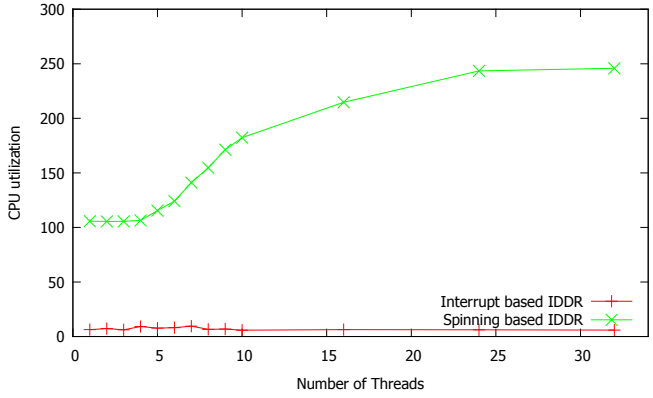
2 cores of the system may be consumed just for spinning. Figures 5.5 and 5.6 show the CPU utilization for the ramdisk and loop device setups. It can be seen that the CPU utilization of the spinning-based implementation ranges between 100% and 250%, which is within the expected range.



(a) Random reads

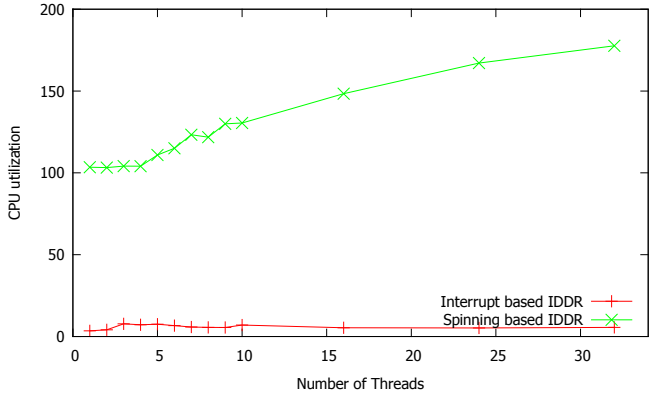


(b) Random writes

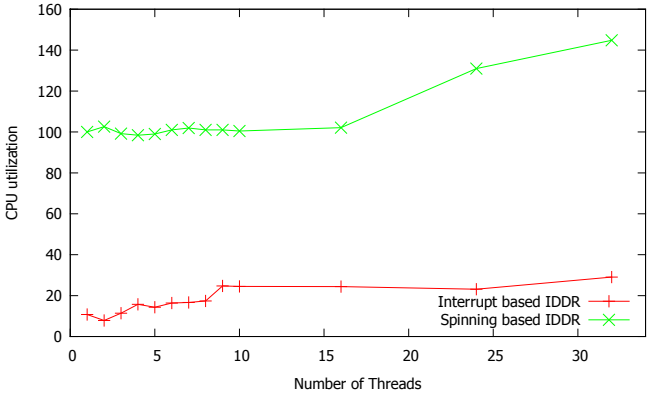


(c) Random reads writes

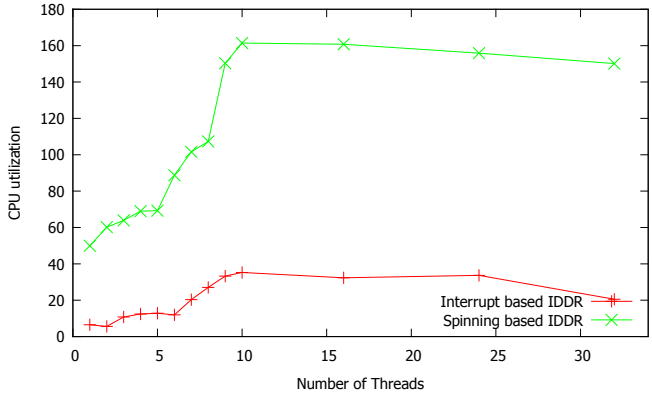
Figure 5.5: Comparison of CPU utilization - ram disk



(a) Random reads

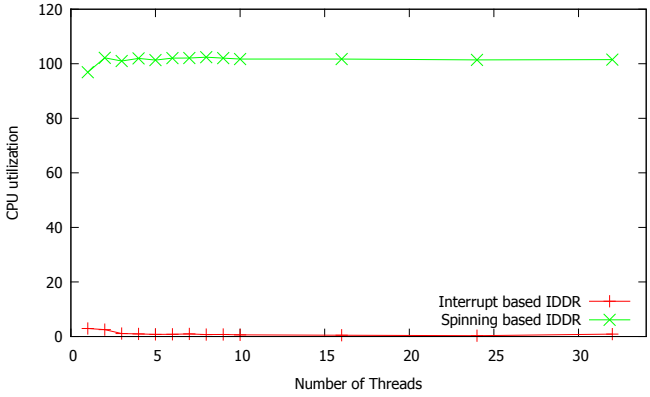


(b) Random writes

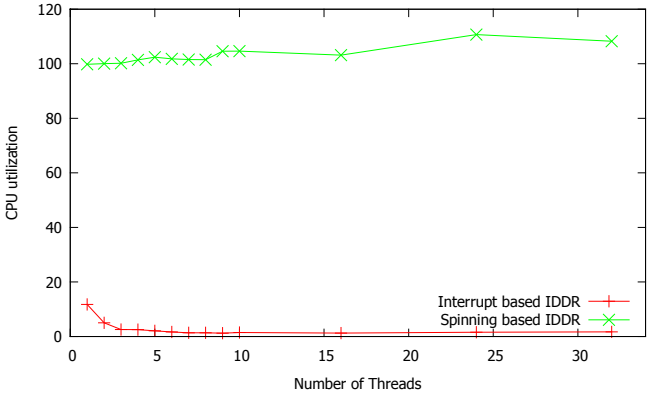


(c) Random reads writes

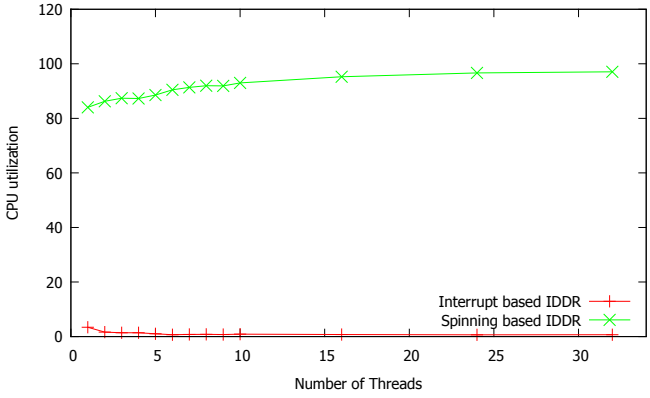
Figure 5.6: Comparison of CPU utilization - loop device



(a) Random reads



(b) Random writes



(c) Random reads writes

Figure 5.7: Comparison of CPU utilization - SATA disk

Chapter 6

Related Work

This chapter discusses work that is closely related to the IDDR system.

Since our work focuses both on improving the reliability of a system through device driver isolation and exploring opportunities to improve their performance, we discuss work related to each of these aspects. Section 6.1 discusses work on improving reliability and Section 6.2 discusses work which concentrates on optimizing interdomain communication.

6.1 Reliability

6.1.1 Driver Protection Approaches

Other researchers have recognized the need for device driver isolation [**?**, **?**]. Common approaches include user-level device drivers, hardware-based driver isolation schemes, and language-based approaches.

Multiple implementations run device drivers in user mode. Even though user mode device drivers allow user level programming and provide good fault isolation between components, they can suffer from poor performance [**?**] and also require rewriting of existing device drivers [**?**].

Microdrivers [**?**] extend the user mode device driver approach by splitting a device driver into two parts. Performance critical operations run inside the kernel and the rest of the driver code runs in a user mode process. Microdrivers can deliver better performance than pure user-level drivers.

Apart from user mode device drivers, some approaches use hardware-based mechanisms inside the kernel to isolate components. Nooks is one example of such approaches [**?**]. Nooks focuses on making device drivers less vulnerable to bugs and malicious exploits within a traditional monolithic kernel. It creates a lightweight protection domain around each device driver. A wrapper layer monitors all interactions between the driver and the kernel, and

protects the kernel from faulty device drivers. Nooks requires device drivers to be modified as their interaction with the kernel changes.

SUD [**?**] runs unmodified Linux device drivers in user space. It emulates IOMMU hardware to achieve this. Running device drivers in user space allows the use of user-level development and debugging tools.

Dune [**?**] is a system that provides an application with direct and safe access to hardware features such as page tables, tagged TLBs, and different levels of privileges. It uses virtualized hardware to isolate applications from each other. Dune delivers hardware interrupts directly to applications. Dune isolates different applications from one another, but it does not directly isolate device drivers.

Virtualization Based Approaches

Fraser et al [**?**] originally proposed isolated driver domains for Xen. IDDR adopts their idea, but provides an entirely separate implementation, along with the optimizations described in Section 1.2.

LeVasseur et. al. [**?**] presents a virtualization based system to reuse unmodified device drivers. In this approach, an unmodified device driver is run with a kernel in a separate virtual machine, isolating it from faults. The main goal of this system is to reuse device drivers across different operating systems.

VirtuOS [**?**] is a solution that allows processes to directly communicate with kernel compo-

nents running in a separate domain at the system call level. VirtuOS exploits virtualization to isolate the components of existing OS kernels in separate virtual machines. These virtual machines directly serve system calls from user processes.

6.1.2 Kernel Designs

Decomposing kernel functionality into separate components can provide better fault containment. Microkernels such as Mach [**?**] and L4 [**?**] are examples of such an approach. Microkernel-based designs include only essential functionalities such as memory management, interprocess communication, scheduling, and low level device drivers inside the kernel. All remaining system components, such as file system and process management, are implemented inside user processes that communicate via message passing [**?**].

Microkernels and hypervisors provide similar abstractions[**?**, **?**]. A Microvisor [**?**] is a microkernel design that allows the execution of multiple virtual machines.

6.2 Interdomain Communication

Related work has also focused on improving interdomain communication.

In Xen VMM, a domain communicates with the privileged domain through the split device driver mechanism [**?**], which incurs the copying and page flipping overheads discussed in Section 3.2. In order to overcome the page flipping performance overhead, Zhang et

al [**?**] proposed a UNIX domain socket like interface called XenSocket, which provides high throughput interdomain communication. The use of XenSockets requires a specific API, so frontend and backend drivers cannot benefit from it transparently.

Fido [**?**] is a shared memory based interdomain communication mechanism. Fido speeds up interdomain communication by reducing data copies in the Xen hypervisor. In contrast, the IDDR system improves the interdomain communication mechanism of split device drivers by avoiding context switches. Fido achieves its goals of improving communication performance by sacrificing some security and protection guarantees.

Chapter 7

Conclusion

In this thesis we presented the Isolated Device Driver (IDDR) system, which we designed and implemented. The IDDR system is an extension to an operating system which provides isolation between a device driver and kernel components by running the device driver in an isolated driver domain. The IDDR system is an independent re-implementation of ideas originally proposed for Xen's isolated driver domains.

Isolated driver domains use Xen's split device driver mechanism, which exploits an interrupt-based approach to interdomain communication. In IDDR, we replaced the interrupt-based approach with a spinning-based approach. This spinning-based approach has the potential to reduce interdomain communication overhead, including the number of context switches. Our evaluation found that we were able to achieve better throughput for a variety of device drivers and workloads.

We also evaluated the CPU utilization since spinning trades CPU cycles for better performance. Spinning may represent a reasonable choice, particularly in environments where I/O performance is paramount and CPU capacity is plentiful.