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

Keywords: Device driver, Virtual machines, domain,

Device driver isolation using virtual machines.

Sushrut Shirole

(ABSTRACT)

In most of today's operating system architectures, a device driver is tightly coupled with the kernel components. In such systems, a malicious or faulty device driver often leads to a system failure, thereby reducing the reliability of the system. Even though a majority of the operating systems provide a protection mechanism at the user level, they do not provide the same level of protection between kernel components. The Isolated Driver Domain is a conceptual framework presented by Xen, which takes advantage of the protection present between the domains and it isolates the device driver and the operating system kernel in separate domains. The Isolated Device Driver (IDDR) implements this framework. Unfortunately, the isolated driver domain solution incurs performance overhead.

Even after many advances in virtualization technologies, current inter-domain communication techniques have significant performance overheads. In this thesis, we propose a solution for performance improvement in inter-domain communication. We implement the proposed solution in the IDDR system.

To evaluate the performance improvement, we implement the prototype based on the block device driver. We isolate and run the block device driver in a separate domain than the

Linux kernel. Our prototype retains the compatibility with current applications and kernel components. Our solution outperforms Xen's isolated driver domain in most scenarios. In other scenarios, its performance matches that of Xen's isolated driver domain.

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 [1].

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 [17, 38]. 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. [29] and Fraser et. al. [19, 5]. 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 [15].

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

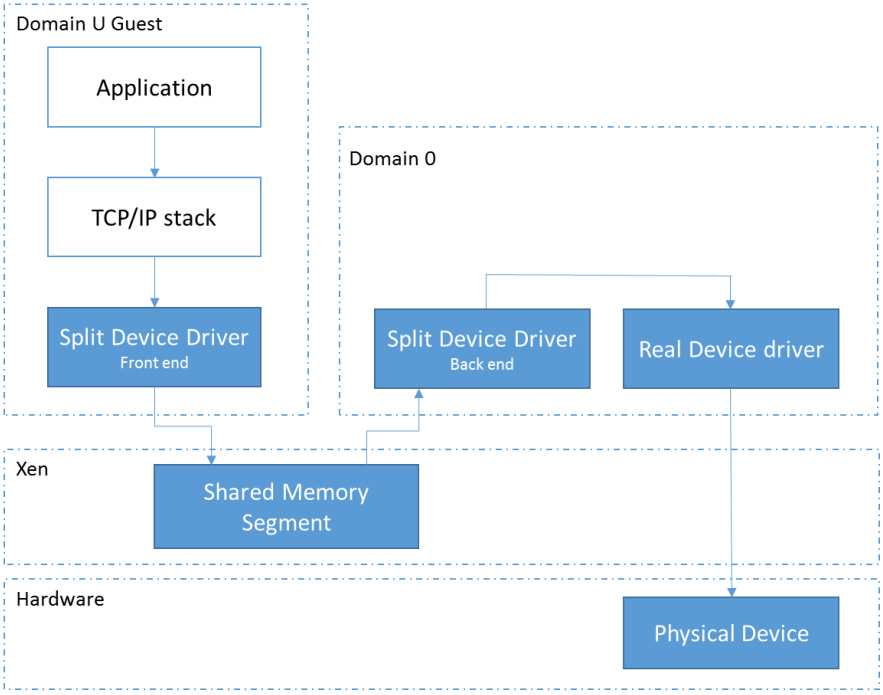


Figure 1.1: Split device driver model

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 [5].

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 [9, 40, 33]. Isolated driver domains follow an interrupt based approach for sending notifications [9] 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 [9]. Such context switches can cause significant overhead [30, 34].

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 [11].

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 [5] 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 [38].

Threads: A process has either a single or multiple threads sharing its address space. Each thread represents a separate flow of control [38].

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: Thread

2.2 Context Switch

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 [38]. 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 [39, 30].

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 [11].

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 Driver

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.

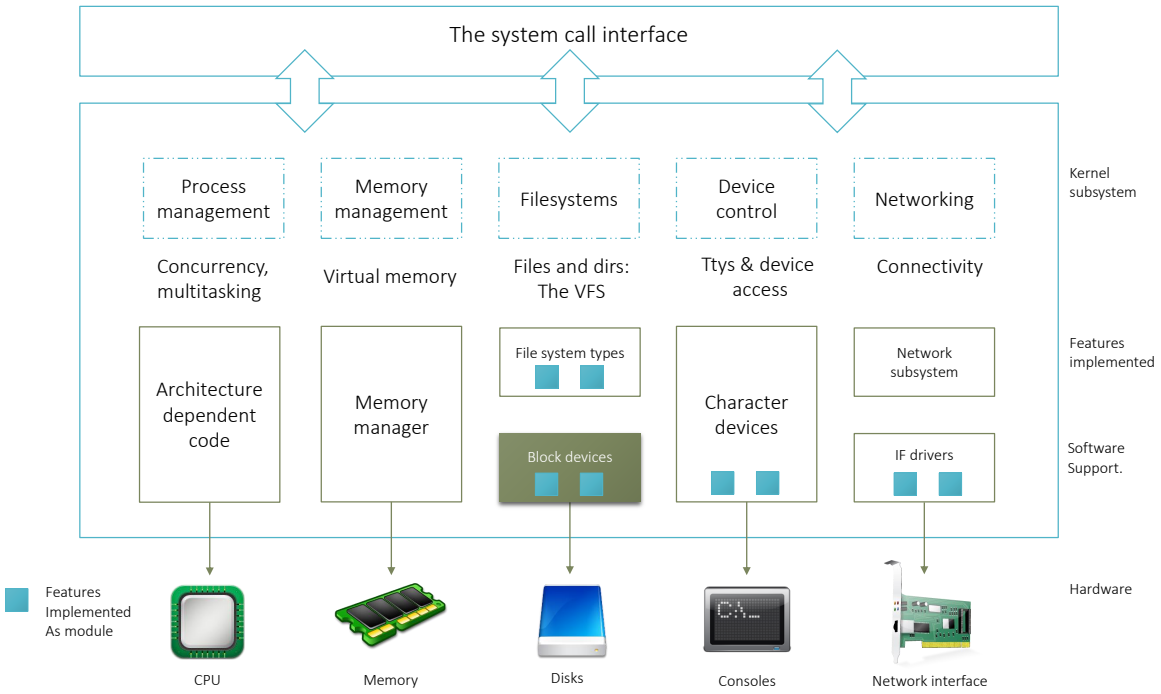


Figure 2.2: Split view of a kernel

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 [16].

Block Device Driver

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 [16].

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 [16], 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 [38, 22].

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

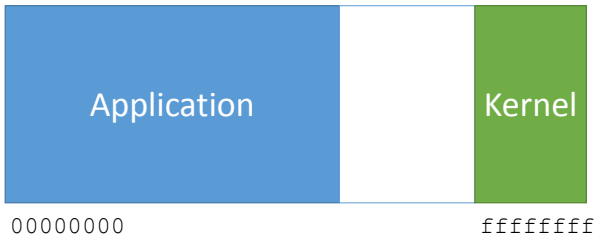


Figure 2.3: Physical memory

blocks called as page frames.

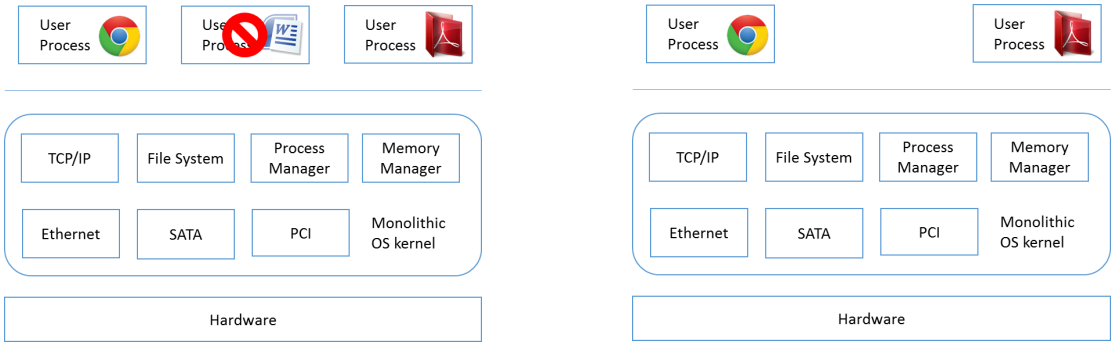
On x86-based systems, the kernel resides in the upper portion of a user application's address space. The application has access to `X Gb` of virtual address space, located at the lower end, whereas the upper `Virtual Memory size - X Gb` are reserved for the kernel. The kernel region is shared between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

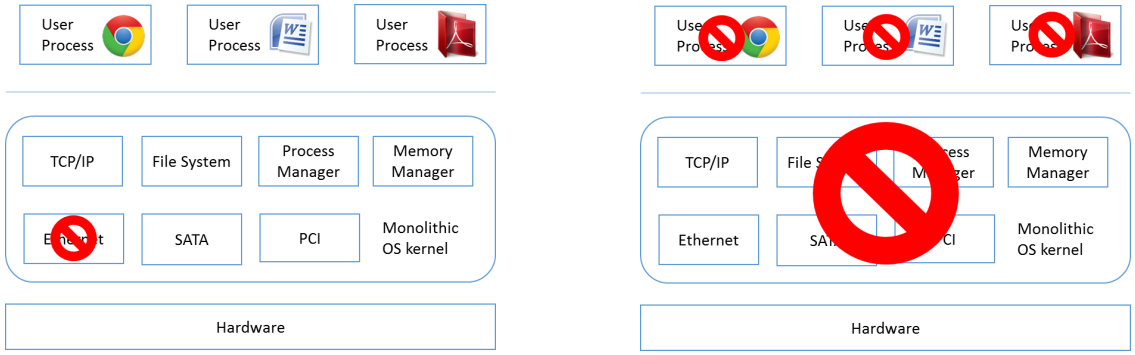


(a) A process encounters a bug (b) Other user processes are not affected

Figure 2.4: User level memory protection

Consider the example shown in Figure 2.6

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

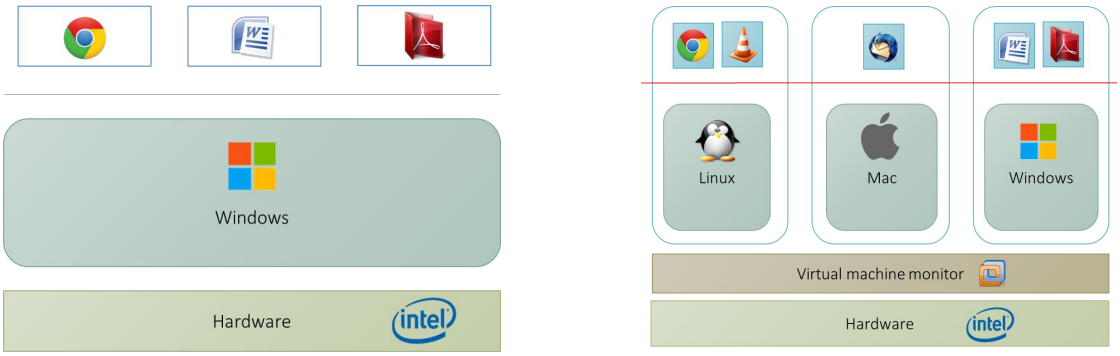
(b) Results in a system crash

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 [38].

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 [18]. It has been shown that virtualization can be utilized to provide better security and robustness for operating systems [19, 29, 37].



(a) Operating System Architecture (b) Virtualization

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

2.6.1 Role of hypervisor

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 [7].

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 [9, 15], KVM [23, 27], VMware ESXi [7] and VirtualBox [14].

There are two types of hypervisors [36]

- 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 [9], VMware ESX/ESXi [7] and Microsoft Hyper-V.

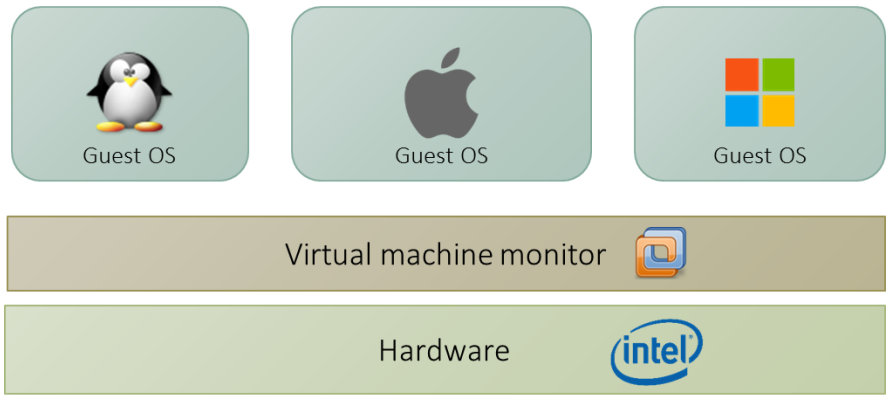


Figure 2.7: Type 1 hypervisors

- 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 [40, 14].

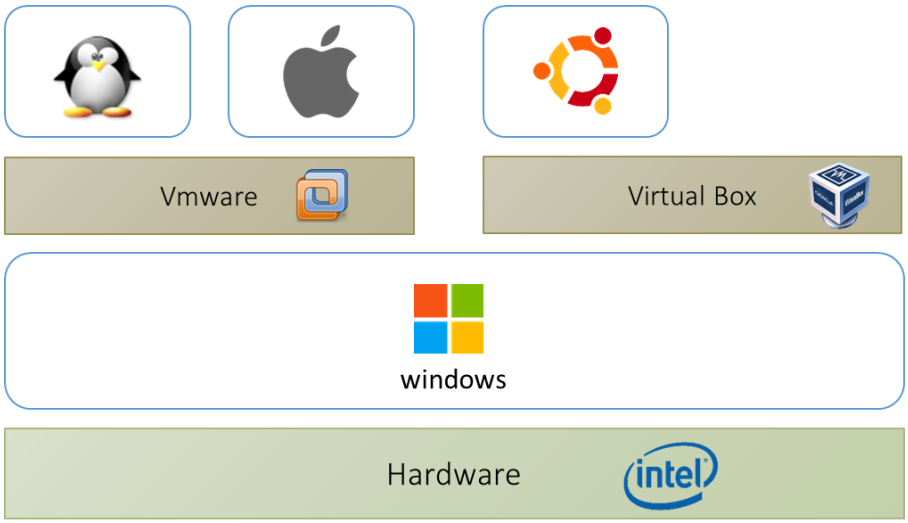


Figure 2.8: Type 2 hypervisors

2.6.2 Xen Hypervisor

Xen [9] is a widely known Type 1 hypervisor that allows the execution of virtual machines in guest domains [26], 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 [15].

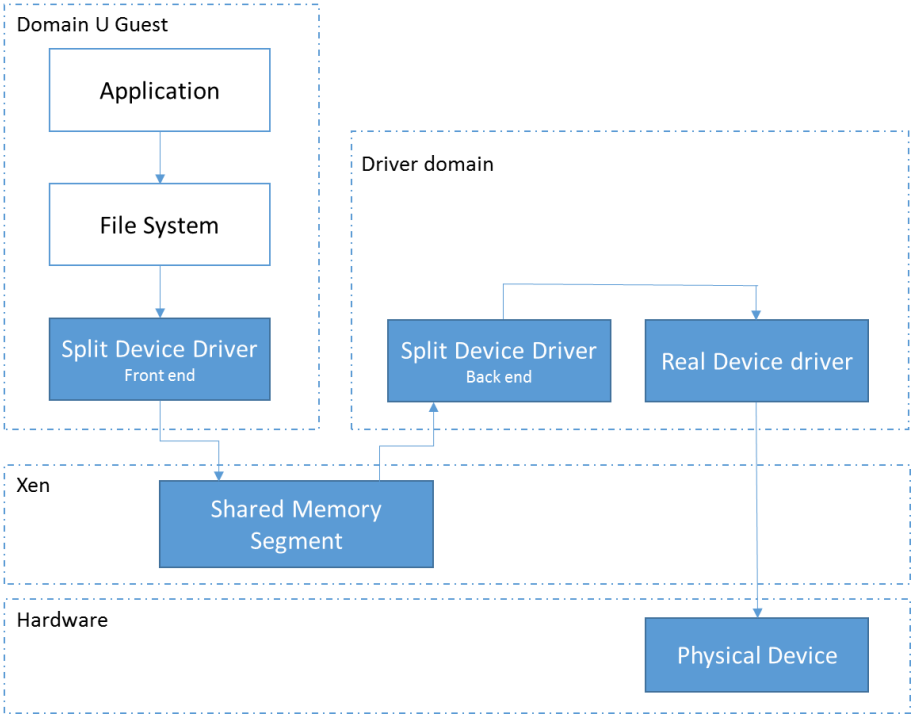


Figure 2.9: Xen split device driver

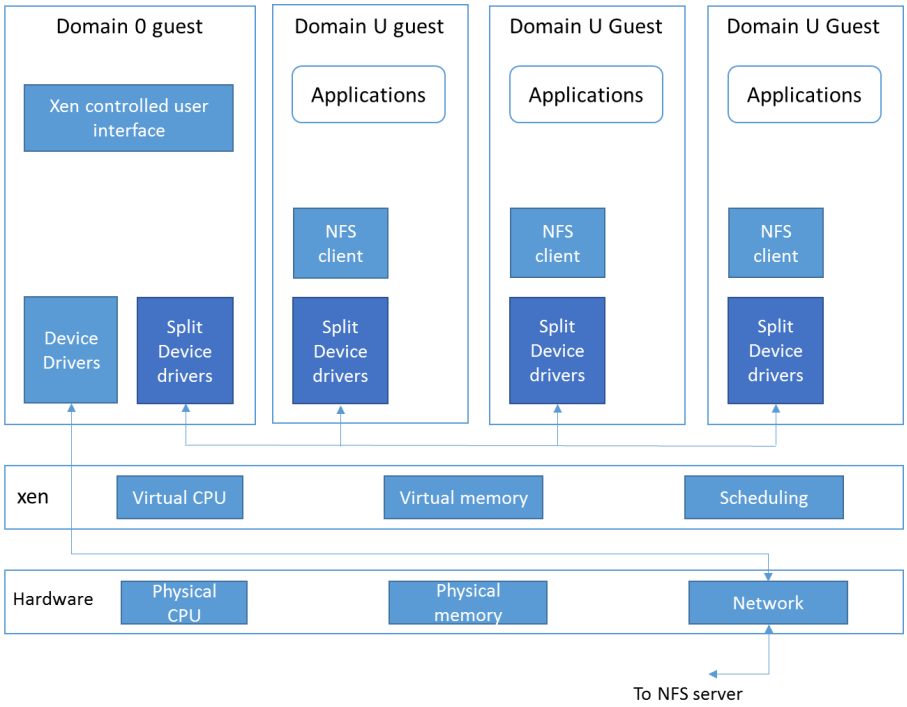


Figure 2.10: Xen

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 [4]. 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.

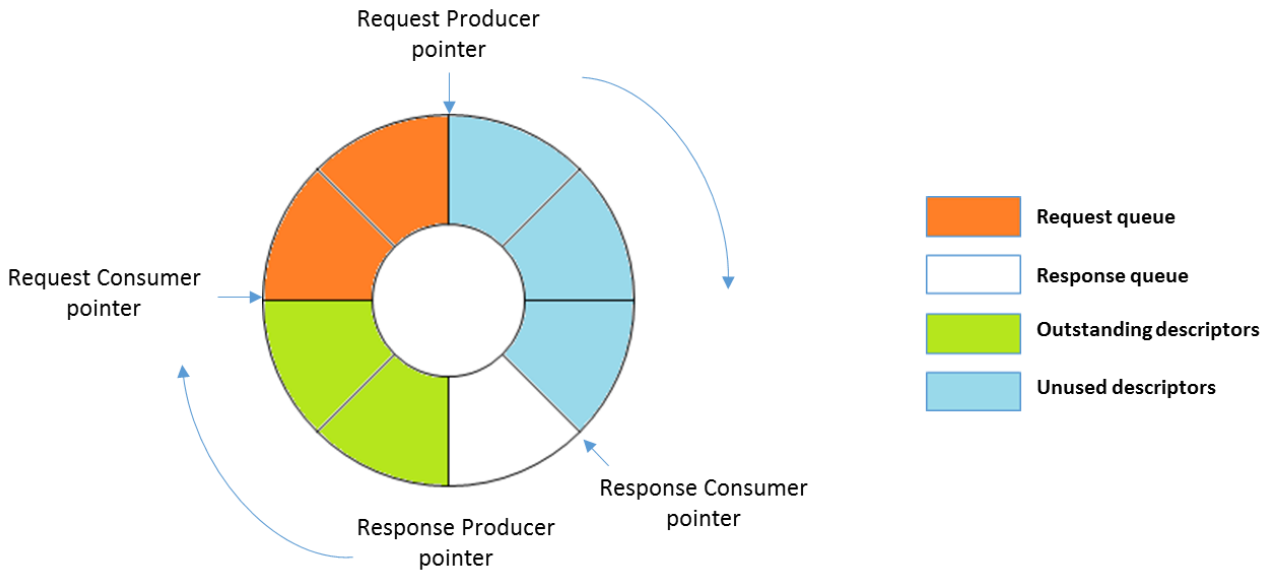


Figure 2.11: Ring I/O buffer

Figure 2.12 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 advancing 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.

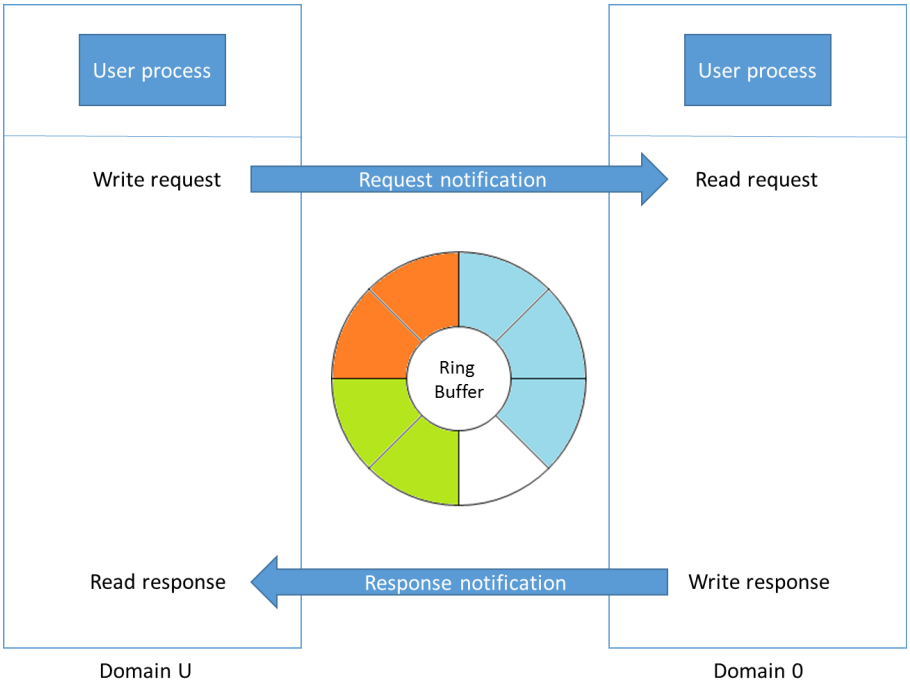


Figure 2.12: Ring I/O buffer

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 [3].

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.[15, 9, 3]

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.

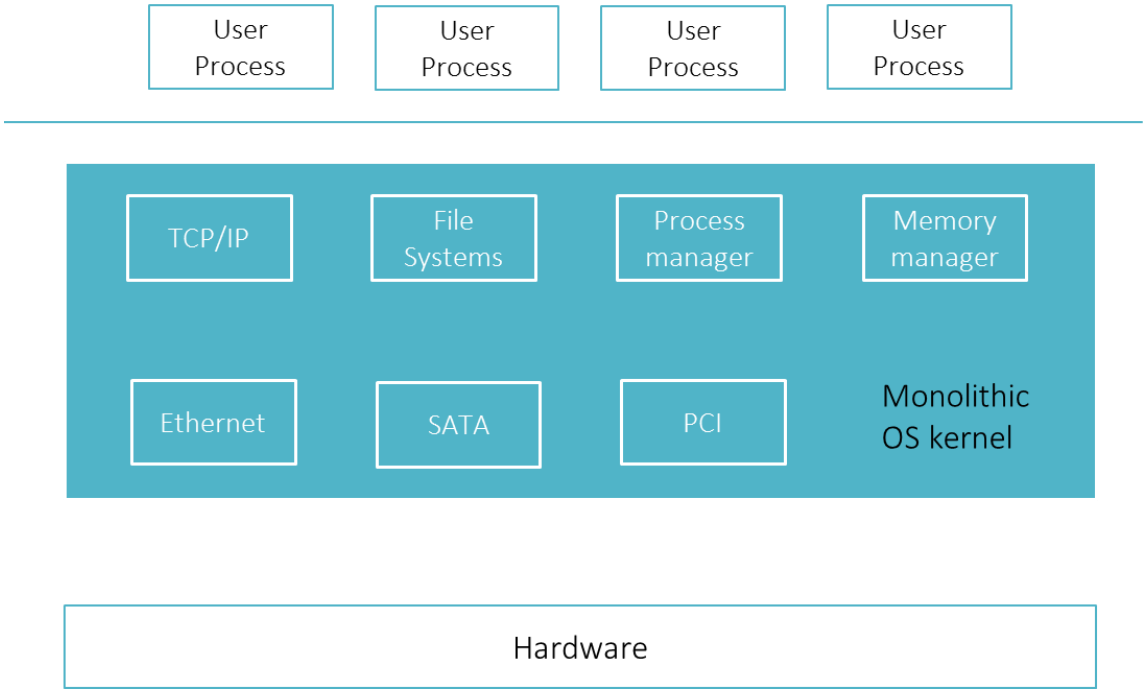


Figure 3.1: Architectural overview of a modern OS

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.

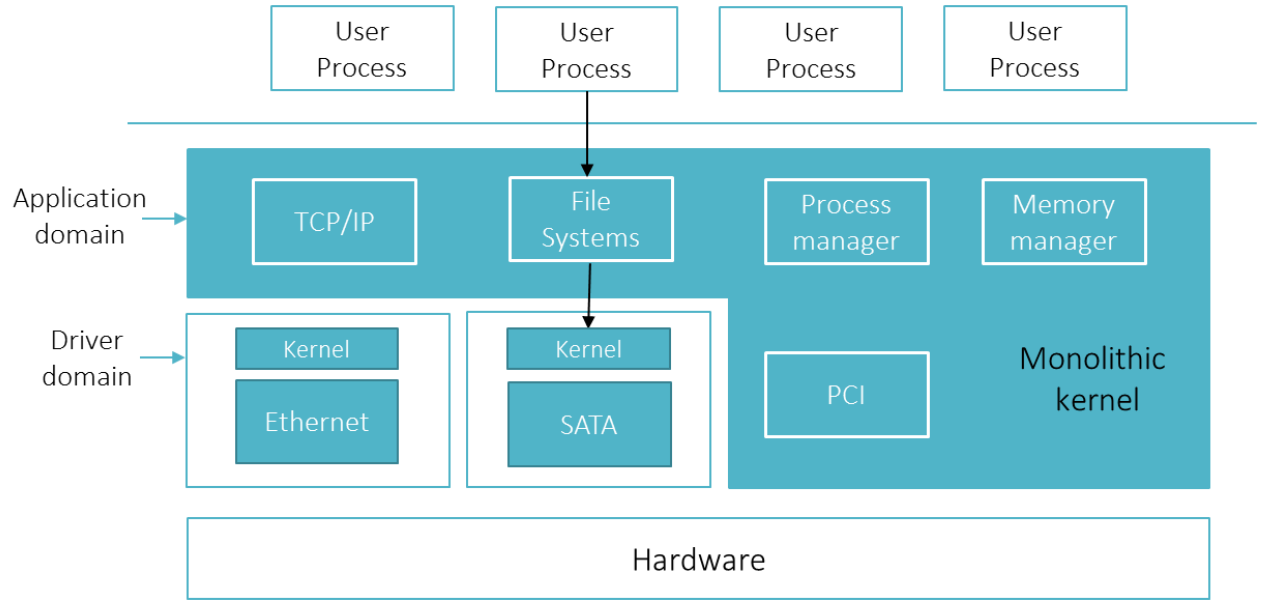


Figure 3.2: Architectural overview of the interrupt based IDDR system

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

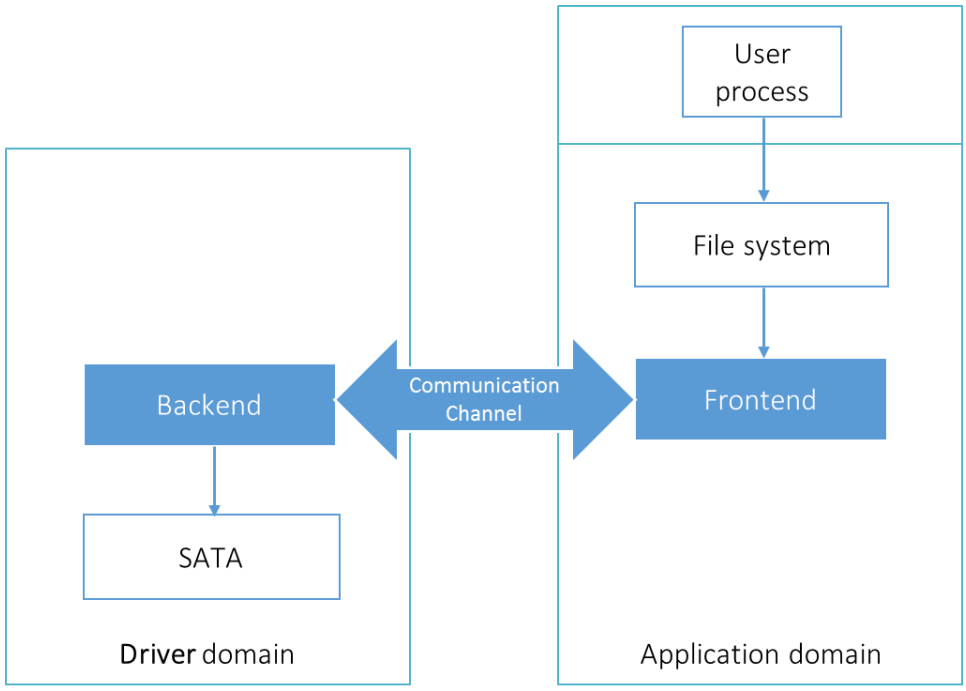


Figure 3.3: System Components

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.

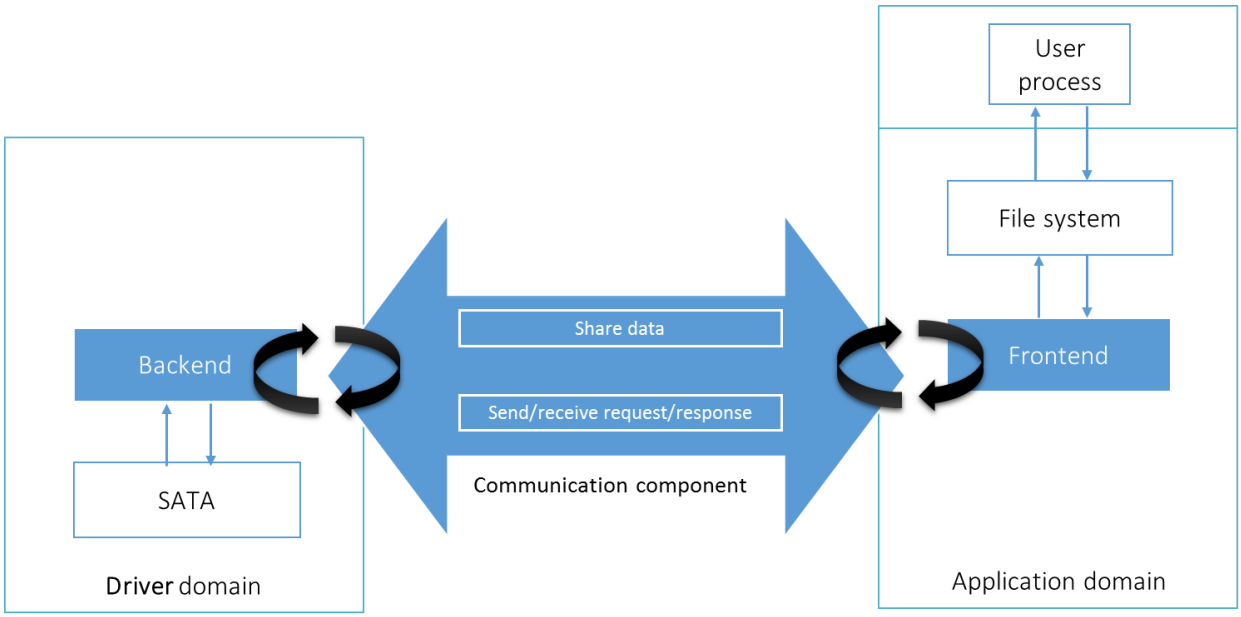


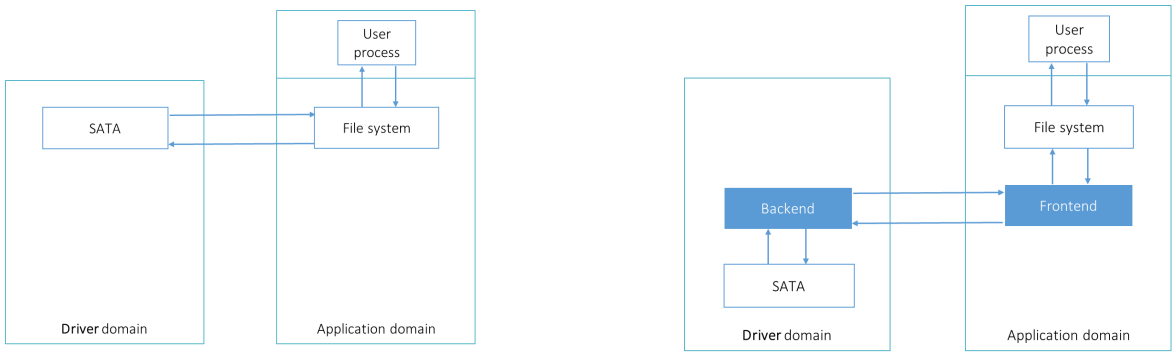
Figure 3.4: Spinning based IDDR system

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.

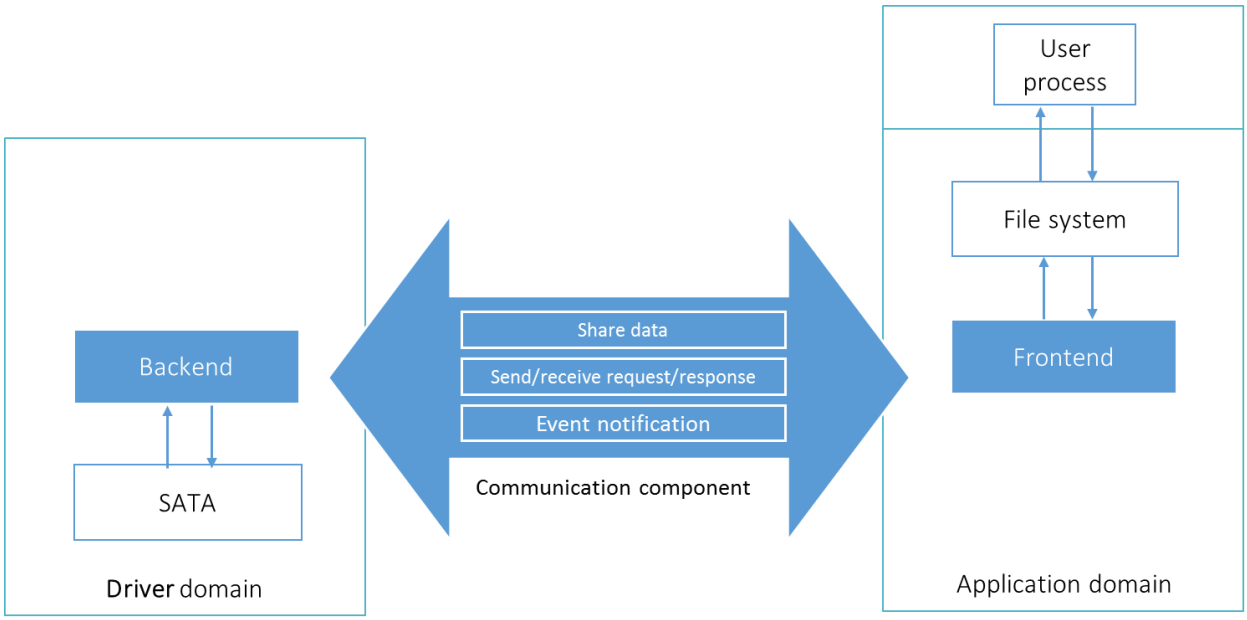


Figure 3.6: Communication Module

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 IDDR system.

4.1 Implementation Overview

We implemented the IDDR system in the Linux 3.5.0 kernel and Xen 4.2.1. The application domain and the driver domain run the same kernel binary. Table 4.1 summarize our implementation efforts in terms of 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.

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

4.2 Implementation

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

4.2.1 Communication Module

This section describes the implementation details of the communication module of interrupt based IDDR system and spinning based IDDR system.

Interrupt based IDDR system

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

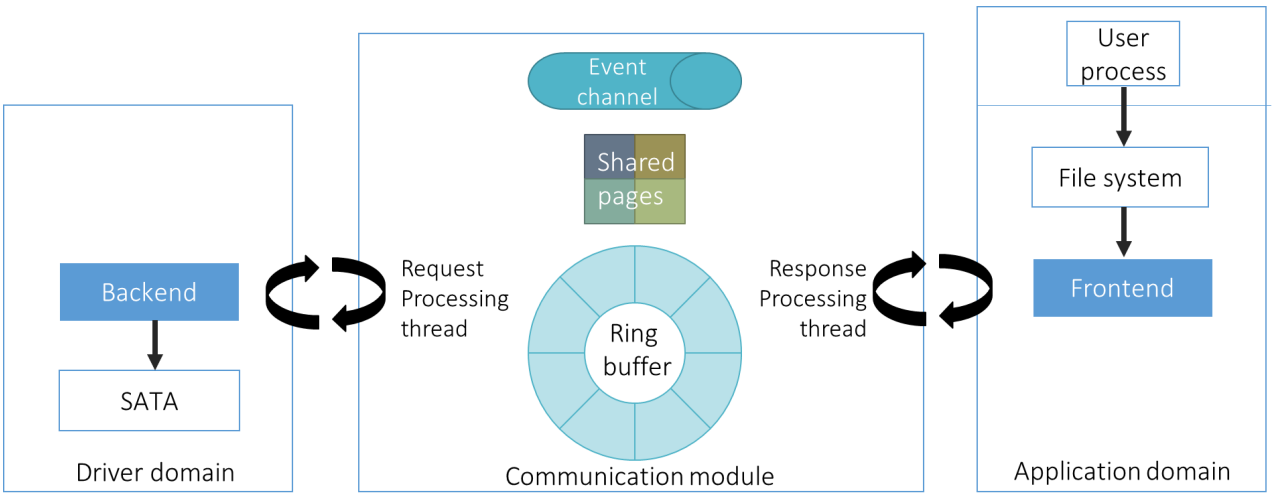


Figure 4.1: Implementation overview of spinning based IDDR system

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

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 ring and the back ring. The IDDR system uses the front ring as the shared request queue and the back ring as the shared response queue. The front ring shares requests and is called as *shared request queue*. The back ring shares responses and is called as *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 for a free space in the shared request queue, and if available, it 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 requests and responses. In order to share data associated with read/write requests/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 a new event channel in the initialization stage of the communication module in the application domain and connect to the same event channel in the initialization stage of the communication module in the driver domain. We attach an interrupt handle routine for the event channel in both the application and driver domain. 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.

Spinning Based IDDR System

This section describes the communication module in 3 parts:

Shared Request and Response Queue: Similar to interrupt based IDDR system, the communication module uses a ring buffer as the shared request and response queue.

Shared Memory for Read/Write Data: Similar to interrupt based IDDR system, the communication module uses a grant table to share an allocated memory between the application domain and the driver domain.

Threads and Event Notification: In order to improve the performance of the IDDR system, we implement the communication module where a thread in the frontend driver spins for the availability of responses, and a thread in the backend driver spins for the availability of requests. In case of unavailability of requests and responses, both threads go to sleep.

The spinning based IDDR system uses an event channel to wake the read request thread sleeping in the application domain. The wake up signal is sent in the form of an event channel notification from the driver domain to the application domain. Similarly, to wake the read response thread sleeping in the driver domain, an event channel notification is sent

from the application domain to the driver domain.

- **Read response thread in the application domain:** In spinning based IDDR system we create a kernel thread called the *read response thread* during an initialization stage of the communication module in the application domain. 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. However, if a response is not available in the shared response queue, after spinning for some time the thread goes into a sleep state. We maintain the status of the thread as `SLEEPING` or `RUNNING` in the shared data structure. We use an atomic variable to save the state of the thread, avoiding race conditions.

  Obviously, a thread shouldn't sleep unless it is assured that somebody else, somewhere, will wake it up. The code doing the waking up must also be able to identify the thread to be able to do its job. We use a Linux data structure called `wait queue` to find the sleeping thread. Wait queue is a list of threads, all waiting for a specific event[38, 11]. We initialize the wait queue for the read response thread during an initialization stage of the communication module in the application domain. The read response thread sleeps in the wait queue, waiting for a flag denoting the availability of the response to be set. The communication module in the driver domain 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 software interrupt through the event channel.

Similar to 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:** In spinning based IDDR system, we create a kernel thread during an initialization stage of the communication module in the driver domain. This new kernel thread is called the *read request thread*. The *read request thread* spins to check if requests are available in the shared request queue. If a request is available, it reads the request from the shared request queue. However, if a request is not available in the shared request queue, the thread goes into a sleep state after spinning for some time (adaptive spinning). Similar to the read response thread, we maintain the status of the thread as `SLEEPING` or `RUNNING` as a atomic variable in the shared data structure.

  We initialize a wait queue for the read request thread during an initialization stage of the communication module in the driver domain. The read request thread sleeps in the wait queue, waiting for a flag denoting availability of the request to be set. The communication module in the application domain checks the status of the read request thread after pushing requests on the shared request queue. If the status is `SLEEPING` then it sends a software interrupt through the event channel.

Similar to interrupt based IDDR system, the driver domain connects to the event channel created by the application domain in the initialization stage of the communication module. 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 request thread if it is sleeping.

4.2.2 Application Domain

Application domain is the domain running user applications and the Linux kernel. In a Linux system, usually a user process sends the read write request to a file system, which sends the read and write request to the block device driver. The block device driver serves the request and sends back a response to the file system, which then sends the response to the user process.

In the IDDR system implementation, a block device runs separately in the driver domain. When a user process sends a request to a file system, the file system needs to forward the request to the driver domain. Like explained in Section 3.4.1, in the IDDR system, a piece of code called the frontend driver forwards the request to the driver domain running the block device driver.

Interrupt based IDDR System

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

1. Provide an interface which appears as a block device to the upper layer in the stack
2. Accept a request from the upper layer
3. Create a corresponding new request which can be understood by the driver domain
4. End the request after reading the response

Implementation details of the frontend driver can be split into 3 stages:

1. Initialization
2. Submit request to the communication module
3. End request

Initialization During the initialization, 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 get queued in this device driver queue.

Dequeue and Submit Request: The frontend driver removes the request submitted to the driver interface and converts the request into a request structure, which is understood by the backend driver. The new 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 also shares the request with the backend driver.

End Request: We maintain a shadow table of all requests which were received in the device driver queue. The shadow table is a table which contains an entry of all the requests received. We implement the shadow table as a circular array of the 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 searches the corresponding request in the shadow table, and ends it.

Spinning Based IDDR System

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

1. Provide an interface for each block device
2. Accept a request from the upper layer
3. Create a corresponding new request which can be understood by the driver domain
4. Spin for a short time while waiting for the response
5. End the request

Implementation details of the frontend driver is split into 4 stages:

1. Initialization
2. Submit request to the communication module

Initialization: Similar to interrupt based IDDR system, during the initialization process the frontend driver creates an interface for each block device.

Dequeue and Submit Request: Similar to interrupt based IDDR system, the frontend driver removes the request submitted to the driver interface and converts it into the request structure with a data pointer pointing to the shared memory. The frontend driver then forwards the newly created request to the communication module, which also shares the request with the backend driver.

4.2.3 Driver Domain

The IDDR system runs a block device driver in the driver domain. Like explained in Section 3.4.2, a piece of code called the backend driver runs in the driver domain, which accepts requests from the application domain and forwards requests to the device driver. Upon receiving a response from the device driver, the backend driver sends back the response to the communication module.

Backend Driver

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 bio request

2. Accept a response from the block device driver

3. Forward the response to the communication module

Implementation details of the backend driver can be split into 2 stages.

1. Convert a request to the bio request.

2. Make a response.

Convert a Request to Bio

The backend driver converts a request that is shared through the communication module into a bio request, so that the block device understands the request. In order to make the bio request, pages from the shared memory are mapped and inserted into the bio structure and required information is copied from the shared request into the bio structure. At the end, the newly created bio request is sent to the lower layer for execution. Once the bio request execution is completed, the system calls a callback function.

Make a Response and Enqueue

Irrespective of the success or failure of execution of a bio request, the backend driver makes a response in the callback function. In this callback function, the result of the bio execution and a request ID is copied into a newly allocated response structure. The request ID is used as an index in the shadow table to map a response and a request. The communication module pushes the response into the shared response queue.

Chapter 5

Evaluation

We use the `Linux kernel 3.5.0` for implementation of application domain and driver domain. We used Arch Linux on `x86_64` platform for the IDDR system testing and performance evaluation. The specifications of the system used for the evaluation is presented in the table 5.1.

5.1 Goals

The goals of the IDDR system evaluation are:

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

   The goal of this thesis is to explore the performance improvement opportunity in the

Table 5.1: Hardware specifications of the system

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

Xen's isolated driver domain and implement it. However, the source code for the isolated driver domain is not available in the open source Xen hypervisor. As a result, we re-implemented the isolated driver domain. We refer to the re-implementation as the interrupt based Isolated Device Driver (IDDR) system. We implemented the spinning based communication channel over the interrupt based IDDR system. We call it as the spinning based IDDR system.

In order to prove the performance improvement of the spinning based IDDR system over the interrupt based IDDR system and the Xen's isolated driver domain, it is necessary to compare the performance of the Xen's isolated driver domain with the interrupt based IDDR system. The Xen's isolated driver domain follows split device

driver architecture. Since the split device driver architecture has gone over decade for the performance testing, the performance comparison with it would prove the solid baseline code of the IDDR system.

We achieve this evaluation goal by comparing the performance of the interrupt based IDDR system with the Xen split device driver. The Comparison shows that the performance of the IDDR system matches the performance of the Xen's isolated driver domain. Hence, proves that our implementation provides a suitable baseline for the performance improvement.

2. **An evaluation of performance improvement:**

   The second goal of the evaluation is to prove that the spinning based communication channel improves the performance of the inter-domain 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. The Comparison shows that the spinning based IDDR system performs better than the interrupt based IDDR system and the Xen's isolated driver domain.

5.2 Methodology

In order to measure the performance of the system, we run performance tests against the variety of block devices. In a Linux system, a loop device is a device that makes a file accessible as a block device. A ramdisk is a block of a memory, which acts as a disk drive. In order to cover a variety of the devices we use block devices such as SATA disk, ramdisk and loop device for the performance testing.

In order to conduct the performance tests, we format the block device with the ext2 file system, and run the fileIO SysBench benchmark [2] on it. SysBench is a multi-threaded benchmark tool for evaluating a system. It evaluates the system performance without installing a database or without setting up complex database benchmarks. SysBench benchmark has different test modes. FileIO is one of the test mode which can be used to produce various file I/O workloads. It can run a specified number of threads by executing all requests in parallel. We run SysBench benchmark in a fileIO test mode to generate 128 files with `1Gb` of total data. We execute random reads, random writes, mix of random read-writes, sequential reads, sequential writes and mix of sequential reads and writes on all the three devices. The block size is kept as `16Kb`. We vary the number of SysBench threads from 1 to 32, to get the throughput of the system under different workload.

5.3 Xen Split Device Driver vs interrupt based IDDR System

As per our first goal of evaluation mentioned in Section 5.1, we compare the interrupt based IDDR with the Xen split device driver.

Experimental Setup

Xen Split Device Driver

We create a ramdisk in the domain 0. The guest domain (domain U) is configured such that the ramdisk uses a split device driver and is available in the guest domain. We do a similar setup for the loop device and the SATA disk. In case of loop device, we create a loop device in the domain 0 and then configure the guest domain to use the loop device as a disk. In case of SATA disk, we configure the guest domain to use SATA disk as a secondary disk. We format and mount the disk in the guest domain with the ext2 file system. SysBench benchmark is run on the mounted partition as explained in section 5.2.

Interrupt Based IDDR System

In a Xen hypervisor, the domain 0 always runs as a paravirtualized guest and in our setup we run the domain U as a HVM guest.

In Xen, paravirtualized guest is a guest which is aware of the VMM and require special ported kernel to run efficiently without emulation or virtual emulated hardware. Paravirtualization does not require virtualization extensions from the host CPU.

On the other hand, fully virtualized or hardware virtual machine (HVM) guest require CPU virtualization extensions such as Intel VT, AMD-V. The Xen uses modified version of Qemu to emulate hardware for HVM guests. CPU virtualization extensions help to boost the performance of the emulation. A fully virtualized guest does not require special kernel. In order to boost the performance, fully virtualized HVM guest uses special paravirtual device drivers and bypasses the emulation for disk and network IO.

A HVM guest is expected to have less syscall overhead and faster memory bandwidth than a PV guest. In the Xen split device driver setup, the backend device driver runs in the domain 0 and the frontend device driver runs in a domain U. In order to compare the Xen split device driver and the interrupt based IDDR system, it is necessary to have the setup similar to the Xen split device driver. We run the backend driver in the domain 0 and the frontend driver in the domain U.

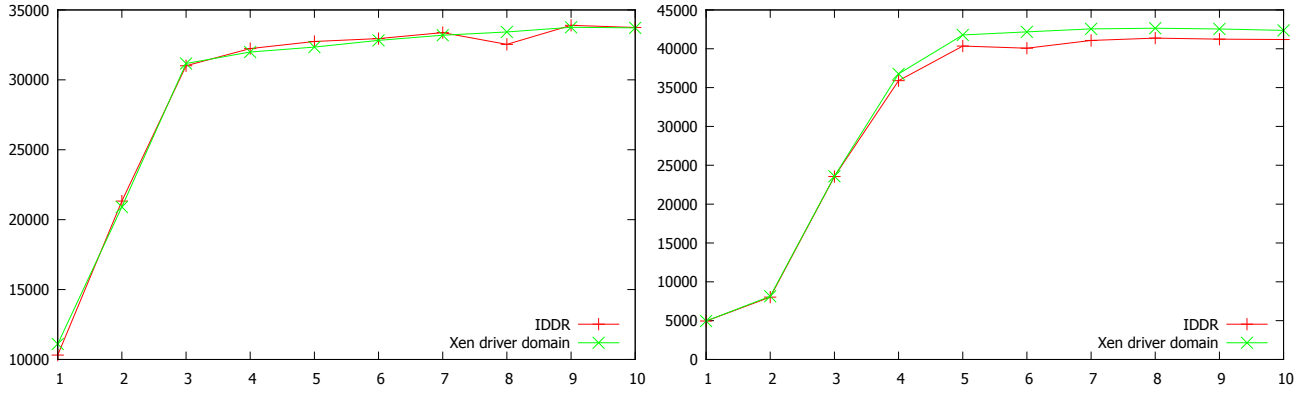
We insert a ramdisk and the interrupt based IDDR system's backend module in the domain 0 and the frontend module in the guest domain (domain U). We format and mount the ramdisk with ext2 file system and run the SysBench benchmark on it.

Similar setup is used for a loop device. We create a loop device in the domain 0 and insert the backend driver in the same domain. We insert the frontend module in the guest domain.

Comparison

We compare the throughput of the Xen split device driver and the interrupt based IDDR system in Figure 5.1a and Figure 5.1b. The Figure 5.1a presents throughput of both systems when data is randomly read from a ramdisk and at the same time written to it. Figure 5.1b presents 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 split device driver. On a ramdisk, the throughput of the interrupt based IDDR system matches that of the Xen split device driver. This proves that our implementation of the isolated driver domain provides a suitable baseline for the performance improvement.



(a) Random reads-writes on a ramdisk

(b) Random reads on a loop device

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. We run the SysBench benchmark in fileIO mode to measure the performance of the system. To compare the performance of both systems, we measure performance of the system by varying the number of SysBench threads. The SysBench benchmark execute random and sequential read write on a ramdisk and loop device.

Experimental Setup

In both systems, the application domain is the domain 0, and the driver domain is a 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 format the ramdisk and mount it with ext2 file system in the application domain.

We measure the performance of both systems on a loop device with similar setup. We create a loop device and insert the backend driver in the driver domain. We insert the frontend driver in the application domain.

Random reads and writes

Comparison : Figure 5.2 and Figure 5.3 compares the throughput of the interrupt based IDDR and the spinning based IDDR system when randomly data is read from a ramdisk and loop device, at the same time data is written on them.

The Figure 5.2a and Figure 5.3a shows that the spinning based IDDR system performs better when data is read from a device randomly.

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

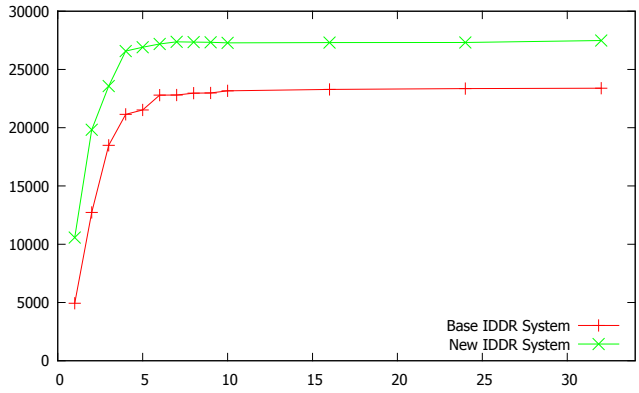
Similarly, the Figure 5.2c and Figure 5.3c compares the mix random read and write performance. In case of mixed reads and writes, most of the time is spent in writing the data. The throughput of the system is dominated by the write performance of it. Figure 5.2c and Figure 5.2b show that curves are similar in both graphs.

Observation : The performance analysis of the interrupt based IDDR system shows that initially throughput of system increases and then it remains constant. We measure the throughput of system with varying number of SysBench threads. When the number of SysBench threads are low, the rate at which data is read and written is low and when number of SysBench threads is high, the rate at which data is read and written is high.

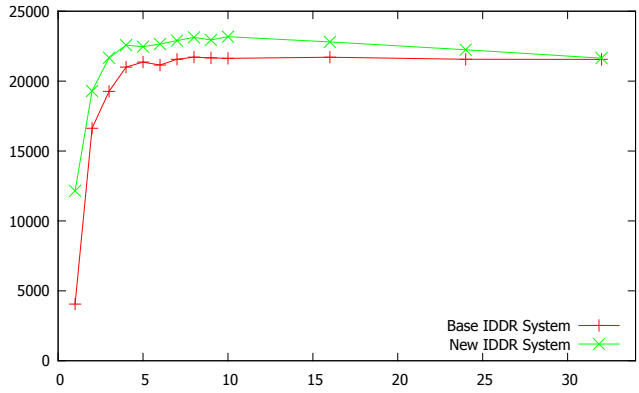
With low data workload, the throughput of a system is bound to be low. The throughput of a system increases as the data workload increase. However, once the bottleneck is hit, the throughput remains constant.

It is not the case with the spinning based IDDR system. Since both the application domain and driver domain spins for the request and responses, when a data workload is low, we observe a more performance gain. When data workload is low, the CPU is idle. Since the frontend and backend driver spin for the request and responses and waste the CPU cycles, which were not in use, the performance of the system increases. So here we see a trade-off between high CPU utilization and high throughput.

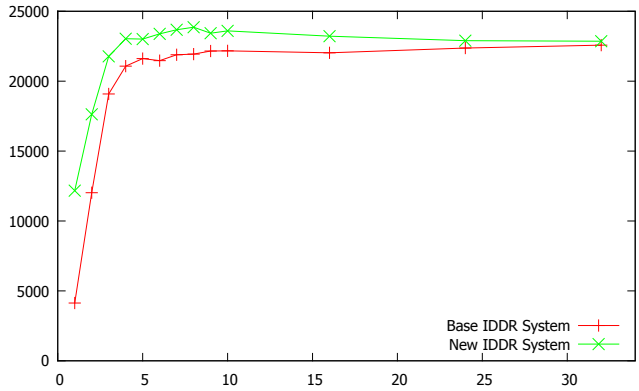
Also when data workload increases, the throughput of the spinning based IDDR system decreases and matches that of the interrupt based IDDR system. Heavy workload denotes the more number of SysBench thread, which denotes the high CPU utilization. Since our system exploits the idle CPU to get the better performance, when the CPU is already under heavy load, the performance of the system decreases.



(a) Random reads

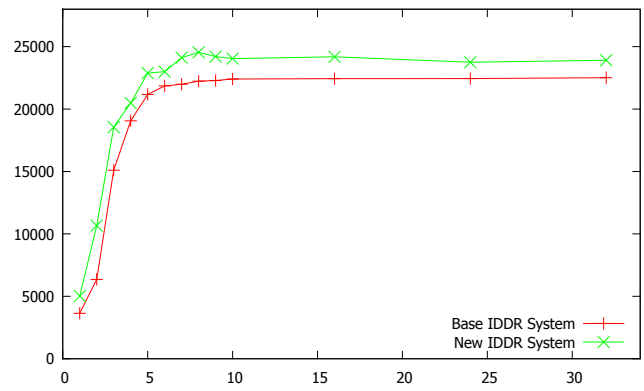


(b) Random writes

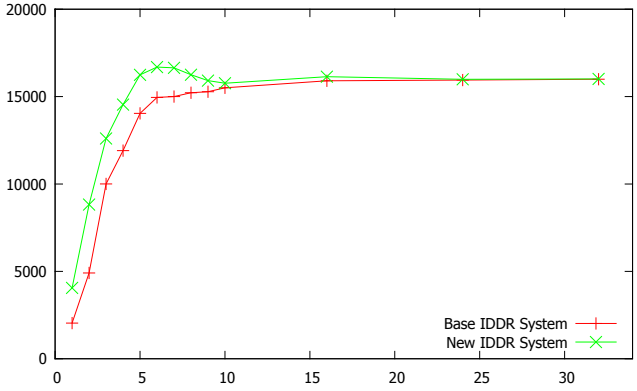


(c) Random reads writes

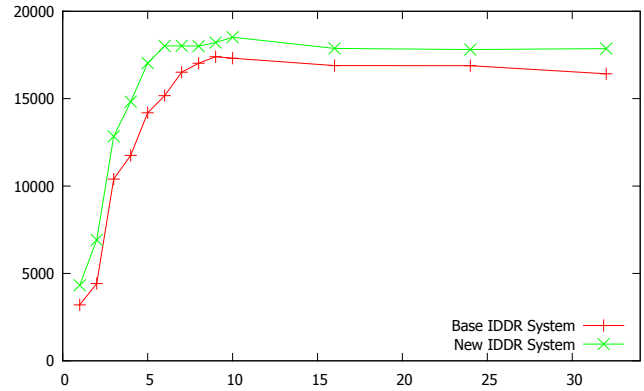
Figure 5.2: Random reads and writes on a Ramdisk



(a) Random reads



(b) Random writes



(c) Random reads writes

Figure 5.3: Random reads and writes on a Loop device

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 [42, 1]. 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 [8] and also require rewriting of existing device drivers [28].

Microdrivers [20] 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 [41]. 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 [12] 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 [10] 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 [19] originally proposed isolated driver domains for Xen. IDDR adopts their idea, but provides an entirely separate implementation, along with the optimizations described in Section **??**.

LeVasseur et. al. [29] 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 [35] 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 [6] and L4 [31] 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 [25] 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 [19], which incurs the copying and page flipping overheads discussed in Section **??**. In order to overcome the page flipping performance overhead, Zhang et

al [43] 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 [13] 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. The IDDR system is an operating system which provides isolation between a device driver and the Linux kernel components by running the device driver in the driver domain. The IDDR system is a re-implementation of the Xen's isolated driver domain.

In Xen's isolated driver domain, a domain communicates with a device driver running in the priviledged domain through a split device driver mechanism. The split device driver follows an interrupt based approach. We replaced the interrupt based approach with the spinning based approach. The spinning based approach avoids the unwanted domain rescheduling for every software interrupt. The IDDR system trades of CPU cycles for the benefit of the performance.

We tested the IDDR system with different block devices, and the experimental evaluation

has shown that the IDDR system performs better by compromizing the CPU utilization.

The IDDR system will be advantagious if used with I/O intensive applications. In the I/O intesive applications, when the workload is low, the CPU utilization is also low. Hence the IDDR system can afford to waste the idle CPU cycles for the benefit of the performance. On the other hand, in case of heavy workload, the CPU utilization is high. Hence if the IDDR system utilizes the CPU cycles in order to improve the performance, it is still acceptable, as those CPU cycles would have been used anyway. However, the IDDR system will hinder the performance of the system, if the system is running CPU intensive applications with an average IO workload.

Bibliography

[1] Coverity's Analysis of the Linux kernel. http://www.coverity.com/library/pdf/coverity_linuxsecurity.pdf.

[2] SysBench 0.4.12 A System Performance Benchmark. http://sysbench.sourceforge.net/.

[3] Xen hypervisor: Grant Table. http://xenbits.xen.org/docs/4.2-testing/misc/grant-tables.txt.

[4] Xen hypervisor: Hypercall. http://wiki.xen.org/wiki/Hypercall.

[5] Xen's Isolated Driver Domain. http://wiki.xen.org/wiki/Driver_Domain.

[6] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–112, 1986.

[7] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4):3–18, December 2010.

[8] François Armand. Give a process to your drivers. In *Proceedings of the EurOpen Autumn 1991 Conference*, pages 16–20. Budapest, 1991.

[9] Paul Barham, Boris Dragovic, Keir Fraser, and et al. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 164–177, Bolton Landing, NY, USA, 2003.

[10] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'12, pages 335–348, Hollywood, CA, USA, 2012.

[11] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 3rd edition, 2005.

[12] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC'10, pages 117–130, Boston, MA, USA, 2010.

[13] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N. Bairavasundaram, Kaladhar Voruganti, and Garth R. Goodson. Fido: Fast inter-virtual-machine

communication for enterprise appliances. In *Proceedings of the 2009 USENIX Annual Technical Conference*, ATC'09, pages 25–25, San Diego, CA, USA, 2009.

[14] Fernando L. Camargos, Gabriel Girard, and Benoit D. Ligneris. Virtualization of Linux servers: a comparative study. In *2008 Ottawa Linux Symsposium*, pages 63–76, Ottawa, Ontario, Canada, 2008.

[15] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, 1st edition, 2007.

[16] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.

[17] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[18] Ulrich Drepper. The cost of virtualization. *ACM Queue*, 6(1):28–35, 2008.

[19] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on-demand IT InfraStructure*, OASIS'04, 2004.

[20] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *Proceedings of*

the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pages 168–178, Seattle, WA, USA, 2008.

[21] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, Massachusetts, USA, 1973.

[22] G. Scott Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, pages 417–429, Atlantic City, NJ, USA, 1972.

[23] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008.

[24] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, HOTOS'05, Santa Fe, NM, 2005.

[25] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems*, APSys'10, pages 19–24, New Delhi, India, 2010.

[26] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the 2003 USENIX Annual Technical Conference*, ATEC'03, pages 71–84, San Antonio, TX, USA, 2003.

[27] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, volume 1, pages 225–230, 2007.

[28] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Gotz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. Userlevel device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[29] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'04, pages 17–30, San Francisco, CA, USA, 2004.

[30] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, San Diego, CA, USA, 2007.

[31] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP'95, pages 237–250, Copper Mountain, CO, USA, 1995.

[32] Daniel A Menascé. Virtualization: Concepts, applications, and performance modeling. In *Int. CMG Conference*, pages 407–414, 2005.

[33] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, pages 2–2, Boston, MA, USA, 2006.

[34] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. *SIGPLAN Not.*, 26(4):75–84, 1991.

[35] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 116–132, Farminton, PA, USA, 2013.

[36] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[37] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID'08, pages 1–20, Cambridge, MA, USA, 2008.

[38] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[39] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'10, pages 1–8, Vancouver, BC, Canada, 2010.

[40] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, volume 15, pages 1–14, Boston, MA, USA, 2001.

[41] Michael M Swift, Brian N Bershad, and Henry M Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, 2005.

[42] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[43] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 184–203, Newport Beach, CA, USA, 2007.