# Performance Optimization 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.

The 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 shows that 53% of the 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, 37]. 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, an 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 increase the reliability of a system were proposed by LeVasseur et. al. [29] and Fraser et. al. [19]. Frazer 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, 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 between these guest domains. They improve the reliability of the system by executing device drivers in separate virtual machines from the kernel. The Xen hypervisor provides a platform to the isolate device drivers from the monolithic kernel. These isolated driver domains are based on the split device driver model exploited by Xen [5].

The Xen virtual machine monitor (VMM) does not include device drivers, instead it is relying

on a dedicated guest domain to provide sudo drivers. The guest typically runs in a privileged domain. This model is called a split device driver model [15]. The split device driver model is shown in the figure 1.1. Xen has a frontend driver in a domain U guest and a backend

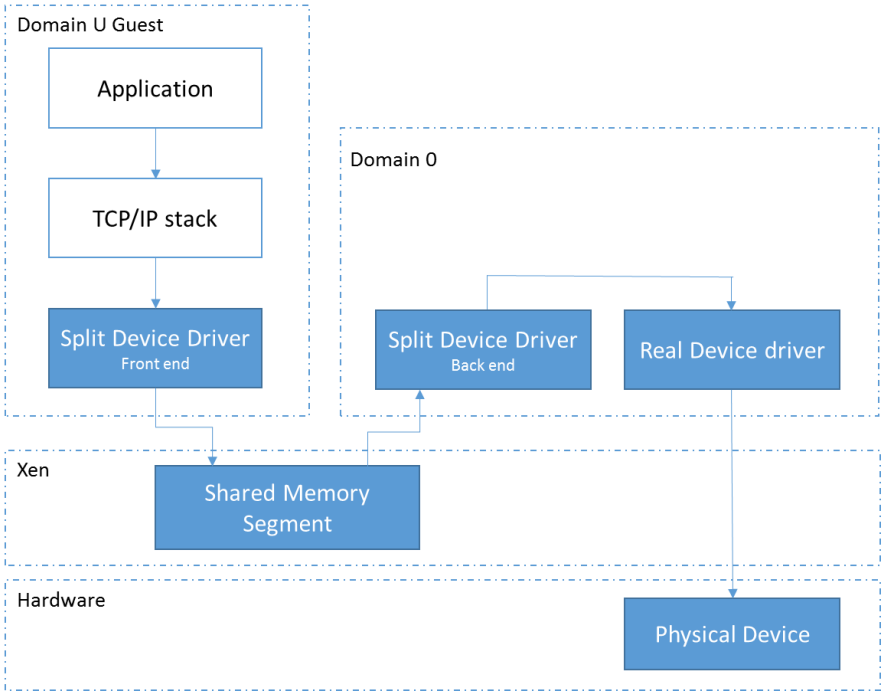



Figure 1.1: Split device driver model

driver in the domain 0. The frontend and the backend driver transfer data between domains over a channel that is provided by the Xen 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].

In the isolated driver domain system, user applications and a kernel are executed in a guest domain, and a device driver is executed in a driver domain. As a result, a device driver is isolated from the Linux 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, 39, 33]. Isolated driver domains follow an interrupt based approach for sending notifications [9]. The frontend and backend 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 system overhead [30, 34].

## 1.2 Proposed Solution

In this thesis, we propose and evaluate an optimization 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, and the frontend driver spins for some amount of time, checking for the responses.

In uniprocessor and multiprocessor environment, a context switch takes a significant amount of time. In a multi-processor environment, it is more efficient for each process to keep its own CPU and spin while waiting for a resource [11]. Since our solution follows a spinning based approach, it performs better than the interrupt based approach used in the original implementation.

The source code of the isolated driver domain is not available in the open source Xen hypervisor code. In this thesis, we re-implement Xen's isolated driver domains, we refer to our implementation as Isolated Device Driver (IDDR). We then implement our spinning based optimization within IDDR.

The performance of the system is evaluated for multiple block device types, including ramdisks, loop devices, and SATA disks. A block device is formatted with a ext2 file system and the IDDR system is evaluated by measuring the performance of the system with SysBench benchmark. The integrity of the system is checked by executing reads and writes on a block device, with and without read ahead, file system tests on the variety of block devices. Our evaluation shows that the performance of the system can be improved by avoiding the context switches in the communication channel. The IDDR system trades off CPU cycles for the better performance. It uses CPU cycles spinning while waiting for requests and responses. Otherwise, these CPU cycles would have been used by an application.

## 1.3 Core Contributions

The core contributions of this project are listed below:

1. Re-implementation of Xen's isolated driver domain - Isolated Device Driver (IDDR).
2. Improvement in the performance of the IDDR by implementing the spinning based communication channel instead of the interrupt based communication channel.

3. A performance comparison of the spinning based IDDR and interrupt based IDDR.

## 1.4 Organization

This section provides the organization and roadmap of the thesis.

1. Chapter 2 provides background on Processes, Threads, Memory Protection, Virtualization, Hypervisor and Inter-domain Communication.

2. Chapter 3 provides an introduction to design of the system to isolate device driver.

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

4. Chapter 5 evaluates the performance of IDDR.

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 a background on operating system terminologies such as Processes, Threads, Memory Protection, Virtualization and Hypervisors.

## 2.1 Processes and Threads

**Process:** A process can be viewed as a program in execution as an abstraction of a processor. Each process has its own address space. [37].

**Threads:** A process has either a single or multiple threads sharing its address space. Each thread represents a separate flow of control [37].

A thread is also called a light-weight process. The implementation of threads and processes

differs between operating systems. Multiple threads can exist within the same process and 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 employs time division multiplexing when sharing a single processor. In time division multiplexing, the processor switches between executing threads, interleaving their execution. 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 multi-processor systems, threads can run simultaneously on multiple processors, each of which may perform time division multiplexing.

Suring a context switch the state of a process is saved, so that its execution can be resumed

from the same point at a later time. The state is determined by the processor and the operating system [37]. The cost of context switch can be divided into direct and indirect cost. The direct cost is the time required to save and restore processor registers, execute the scheduler code, flush TLB entries and flush pipeline. The indirect cost is the time spent due to processor pollution [38, 30].

## 2.3 Spinlocks

In uniprocessor and multiprocessor environment, a context switch takes a significant amount of time. In a multi-processor environment, it is more efficient for each process to keep its own CPU and spin while waiting for a resource [11].

A spinlock is a locking mechanism designed to work in a multi-processing environment. A spinlock causes a thread that is trying to acquire lock to spin if the lock is not available [11].

**Adaptive Spinning** is a spinlock optimization technique. In the adaptive spinning technique, the duration of spinning is determined by 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 unnecessary conditions.

## 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. For a system call requested by a user program, a driver issues commands to the device. After execution, the device sends data back to the driver. The driver may invoke routines in the original calling program after receiving the data. The Linux kernel distinguishes between three device types: character devices, block devices and network interfaces.

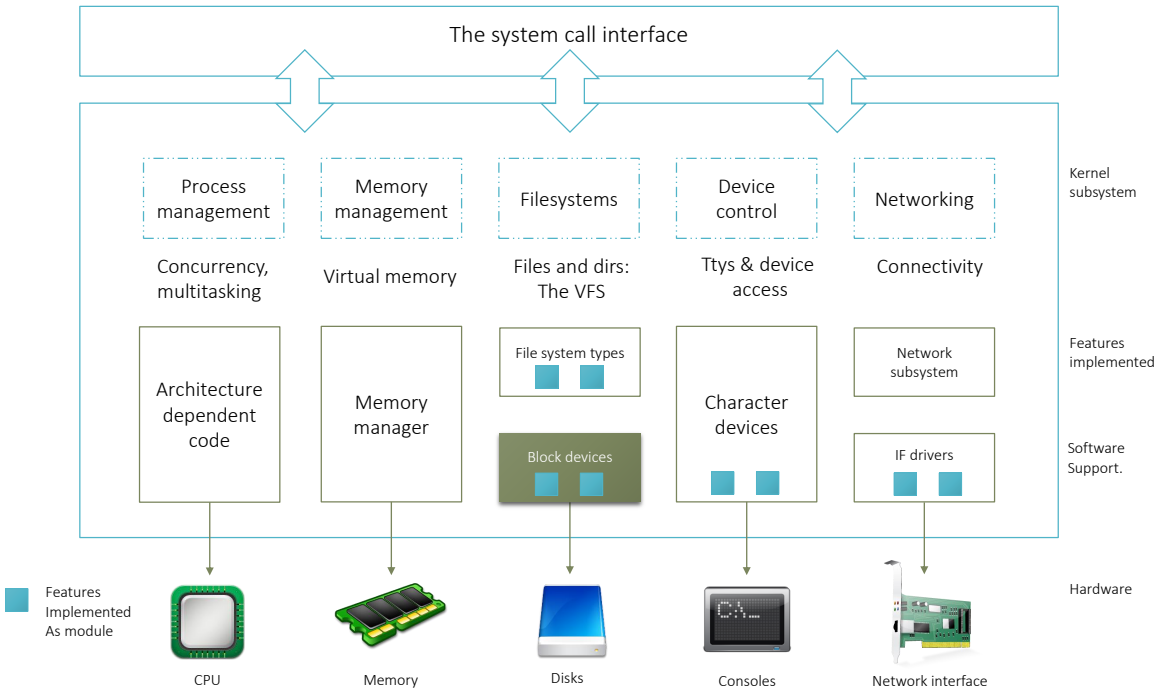



Figure 2.2: Split view of a kernel

**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 like the loopback interface.

**Block Devices:** Unlike character devices, block devices are accessed as blocks of data. In most unix implementations, a block device can only handle I/O operations that transfer one or more whole blocks. Linux, instead, allows the application to read and write any number of bytes on a block device. As a result, 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, write requests. In order to initialize a request queue, a spinlock and a request function pointer is required. Request function is considered as a 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 File systems. A block device driver calls its request function after receiving a new request. The

request function removes all requests from the head of the request queue and sends them to the block device for execution. The Linux kernel acquires a spinlock before execution of the request function and releases it after completing execution. As a result, a request function runs in an atomic context [16].

A request is a linked list of **bio structures**. A bio structure contains all the information required to execute a read and 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 adds the received bio structure into existing request [16].

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 requires access to the adjacent sectors on the disk. However, it never combines a read and write request together.

## 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 [37, 22]. Subsection 2.5.1 and subsection 2.5.2 explain the policies implemented at kernel level and user level.

### 2.5.1 User Level

Typically in a monolithic kernel, the lowest `X Gb` of memory is reserved for user processes. The upper `Virtual Memory size - X Gb` is reserved for the kernel. The kernel puts its private data structures in the upper memory and always accesses them at the same virtual address.

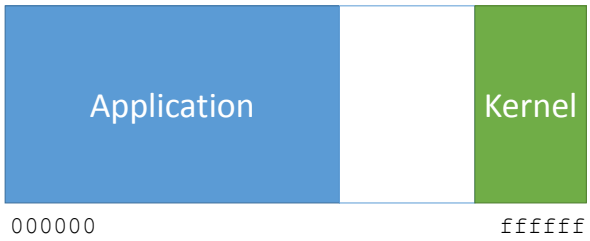



Figure 2.3: Physical memory

At the user space, each application runs as a separate process. Each process is associated with an address space and believes that it owns the entire memory, starting with the virtual address 0. However, a translation table translates every memory reference by these processes from virtual to physical addresses. The translation table maintains <`base, bound`> entries. If a process tries to access virtual address that is outside the `base + bound` address, then an error is reported by the operating system, otherwise the physical address `base + virtual address` is returned. This allows multiple processes to be run in the memory with protection. Since address translation provides protection, a process can not access addresses outside its

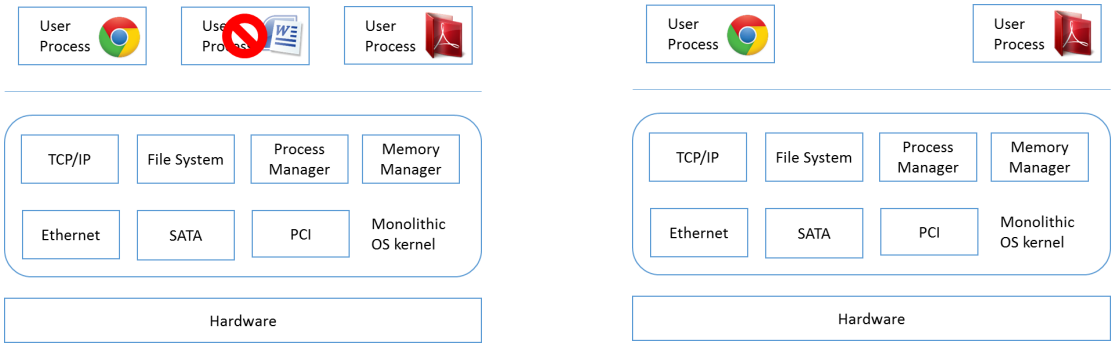
address space.

Consider an example in Figure 2.4.

1. The system is running 3 different processes in a user space.

2. One of the processes hits a bug and tries to corrupt the memory outside the address space.

3. Access to the address is restricted by the memory protection mechanism.




(a) A process hits a bug (b) Intact system

Figure 2.4: User level memory protection

### 2.5.2 Kernel Level

Kernel reserves upper `Virtual memory size - X Gb` of virtual memory for its internal use. The page table entries of this region are marked as protected so that pages are not visible or modifiable in the user mode. This reserved region is divided into two regions. First region

contains page table references to every page in the system. It is used to do translations of addresses from physical to virtual when the kernel code is executed. The core of the kernel and all the pages allocated by page allocator lies in this region. The other region of the kernel memory is used by the memory allocator, the allocated memory is mapped by kernel modules. Since an operating system maps physical addresses directly, kernel components do not have memory protection similar to that of the user space. At kernel level any code running at `CPL 0` can access the kernel memory, hence a kernel component can access, and potentially, corrupt the kernel data structures.

Consider an example shown in the Figure 2.6.

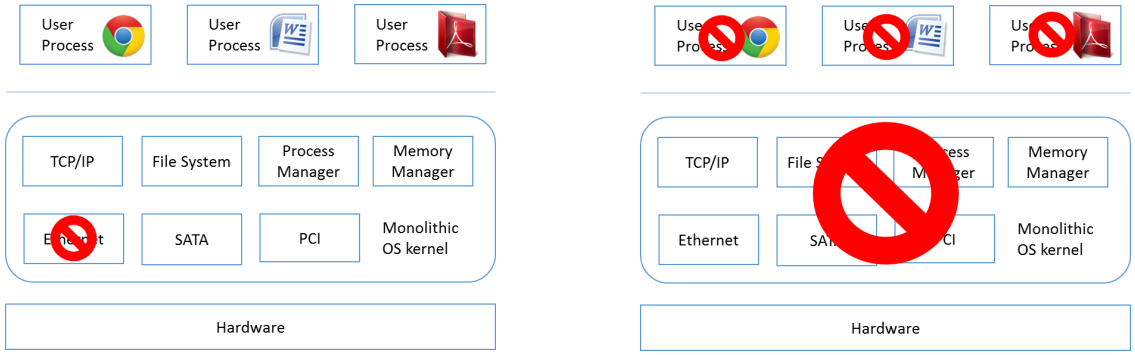
1. The system runs 3 different processes in the user space and has different kernel components running in the kernel space.

2. The network driver hits a bug, and corrupts the kernel data structure. The corruption might lead to a system crash.

## 2.6 Virtualization

Virtualization is the act of creating a virtual version of a hardware platform, storage device, or computer network resource etc. In an operating system virtualization, the software allows a hardware to run multiple operating system images at the same time.

Virtualization has the capability to share the underlying hardware resources and still provide




(a) A kernel component hits a bug

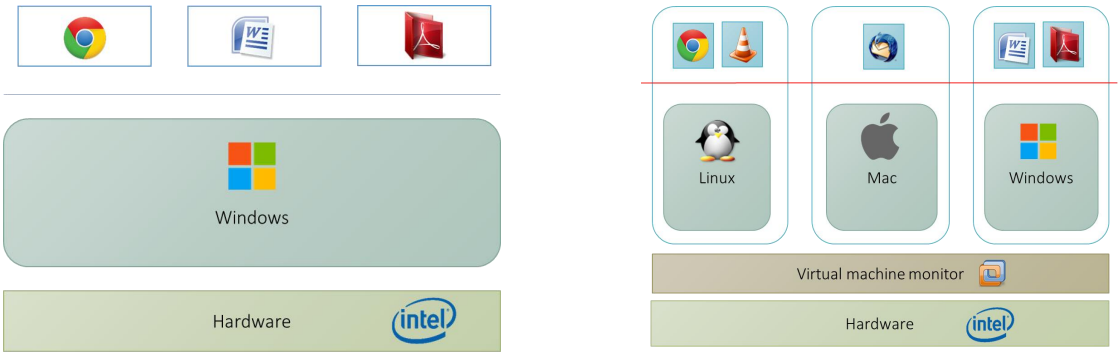
(b) Results in a system crash

Figure 2.5: Kernel level memory protection

an isolated environment to each operating system. In virtualization, each operating system runs independently from the other on its own virtual processors. Because of this isolation the failures in an operating system are contained. Virtualization is implemented in many different ways. It can be implemented either with or without hardware support. Also 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, 36].

### 2.6.1 Hypervisor

Hypervisor is a piece of computer software, firmware or hardware that creates and runs virtual machines. Operating system virtualization is achieved by inserting a hypervisor between the guest operating system and the underlying hardware. Most of the literature presents hypervisor synonymous to a virtual machine monitor (VMM). While, VMM is a




(a) Operating System Architecture (b) Virtualization

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

software layer specifically responsible for virtualizing a given architecture, a hypervisor is an operating system with a VMM. The 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. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of operating systems share the virtualized hardware resources. Among widely known hypervisors are Xen [9, 15], KVM [23, 27], VMware ESX [7] and VirtualBox [14].

There are two types of hypervisors [21]

- Type 1 hypervisors are also called native hypervisors or bare metal hypervisors. Type 1 hypervisors run directly on the host's hardware to control the hardware and to manage

guest operating systems. A guest operating-system, thus, runs on another level above the hypervisor. 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.

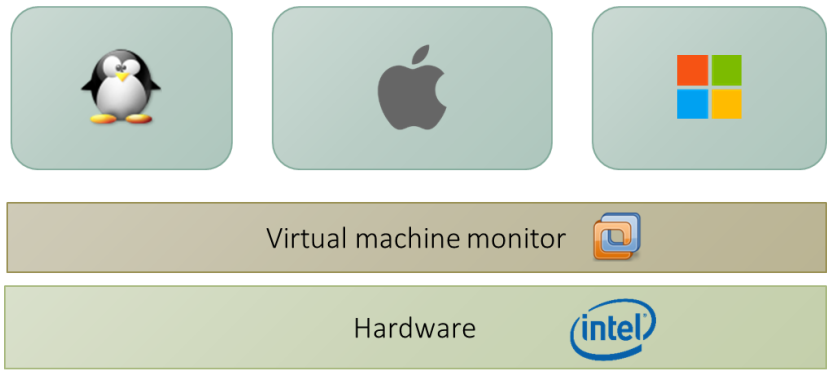



Figure 2.7: Type 1 hypervisor

- Type 2 hypervisors are also called hosted hypervisors. Type 2 hypervisors run within a conventional operating-system environment. Type 2 hypervisors run at a distinct second software level, whereas guest operating systems run at the third level above hardware. VMware Workstation and VirtualBox are some of the examples of Type 2 hypervisors [39, 14].

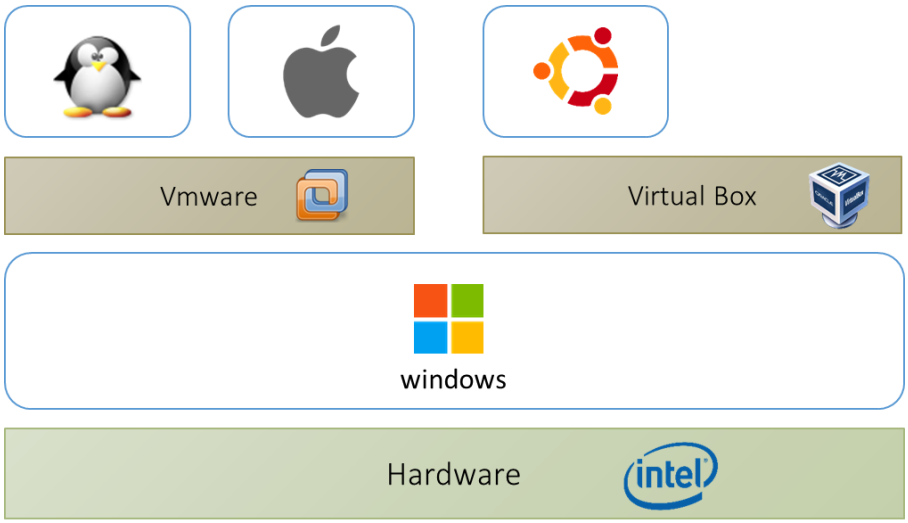



Figure 2.8: Type 2 hypervisor

### 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]. Figure 2.8 represents a diagram showing the different layers of a Type 1 hypervisor system. The hypervisor itself forms the lowest layer, which consists of the hypervisor kernel and the virtual machine monitors. The kernel has direct access to the hardware and is responsible for resource allocation, resource scheduling and resource sharing. A hypervisor is a layer responsible for virtualizing and providing resources to a given operating system.

The purpose of a hypervisor is to allow guest operating systems to be run. Xen runs guest operating systems in environments known as domains. `Domain 0` is the first guest to run,

and has elevated privileges. Xen loads a `domain 0` guest kernel during boot. Other unprivileged domains are called `domain U`. The Xen hypervisor does not include device drivers. Device management is included in privileged `domain 0`. `Domain 0` uses the device drivers present in the guest operating system. However, `domain U` accesses devices using a split device driver architecture. In the split device driver architecture 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 on the physical device. First, it travels through the file system as it would normally. However, at the end of the stack the normal block device driver does not exist. Instead, a simple piece of code called the frontend puts the data into the shared memory. The other half of the split device driver called the backend, running in the `domain 0` guest, reads the data from the buffer and sends it way down to the real device driver. The data is written on the actual physical device. In conclusion, split device driver can be explained as a way to move data from the `domain U` guests to the `domain 0` guest, usually using ring buffers in shared memory [15].

Xen provides an inter-domain memory sharing API accessed through the guest kernel extensions and an interrupt-based inter-domain signaling facility called event channels to implement the efficient inter-domain communication. Split drivers use memory sharing APIs to implement I/O device ring buffers to exchange data across domains.

In Xen's isolated driver domain implementation, Xen uses shared I/O ring buffers and event channel [9, 35].

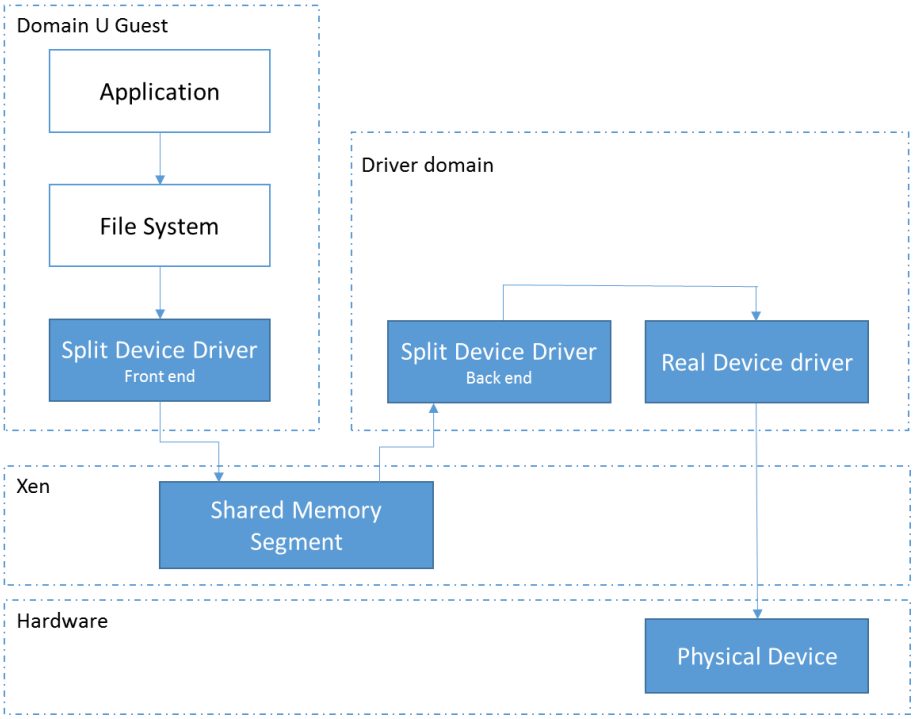



Figure 2.9: Xen split device driver

### Hypercalls and Events

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 the hypercalls to request privileged operations like updating pagetables.

An event channel is to the Xen hypervisor as a hardware interrupt is to the 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

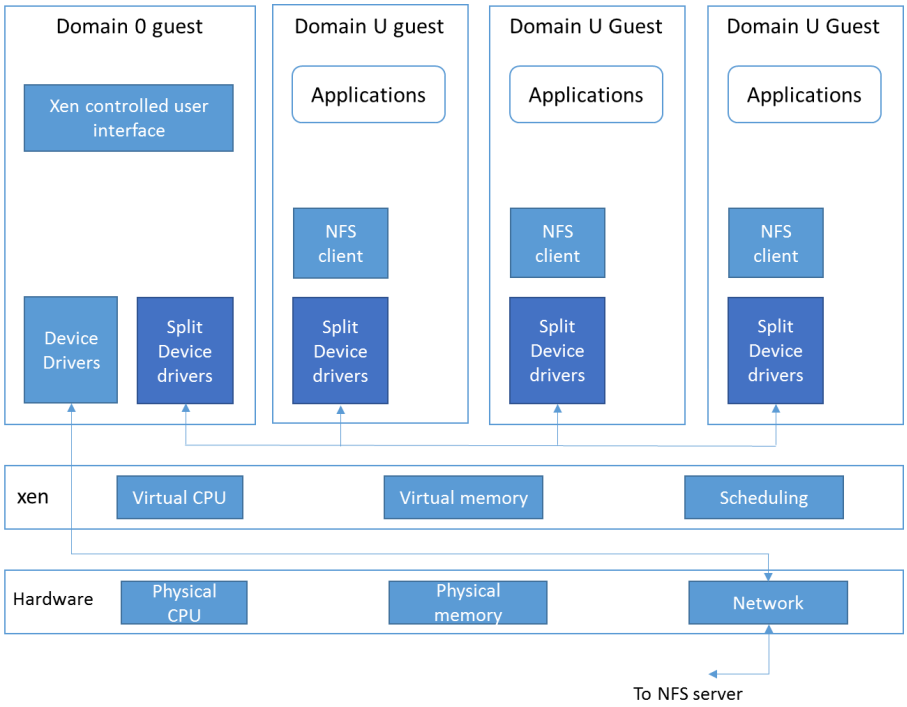



Figure 2.10: Xen

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 the a virtual disk request has completed.

### Data Transfer: I/O Rings

Hypervisor introduces an additional layer between guest OS and I/O devices. Xen provides a data transfer mechanism that allows data to move vertically through the system with minimum overhead.

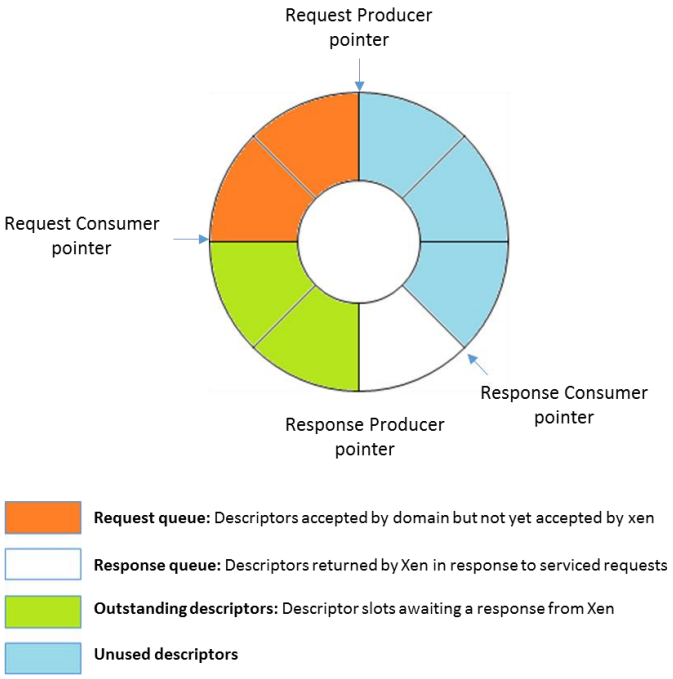



Figure 2.11: Ring I/O buffer

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. However, I/O data buffers are allocated separately by the guest OS and is indirectly referenced by these I/O descriptors. Access to an I/O ring is based around two pairs of producer-consumer pointers.

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

2. Request consumer pointer: The Xen hypervisor removes requests pointed by a request producer pointer. These requests are removed by advancing a request consumer pointer.

3. Response producer pointer: The Xen hypervisor places responses on a ring by advancing a response producer pointer.

4. Response consumer pointer: A domain removes responses pointed by a request producer pointer. These responses are removed by advancing a response consumer pointer.

The requests are not required to be processed in a specific order. I/O rings are generic to support different device paradigms. For example, a set of `requests` can provide buffers for read data of virtual disks; subsequent `responses` then signal the arrival of data into these buffers.

The notification is not sent for production of each request and response. A domain can en-queue multiple requests and responses before notifying the other domain. This allows each domain to trade-off between latency and throughput.

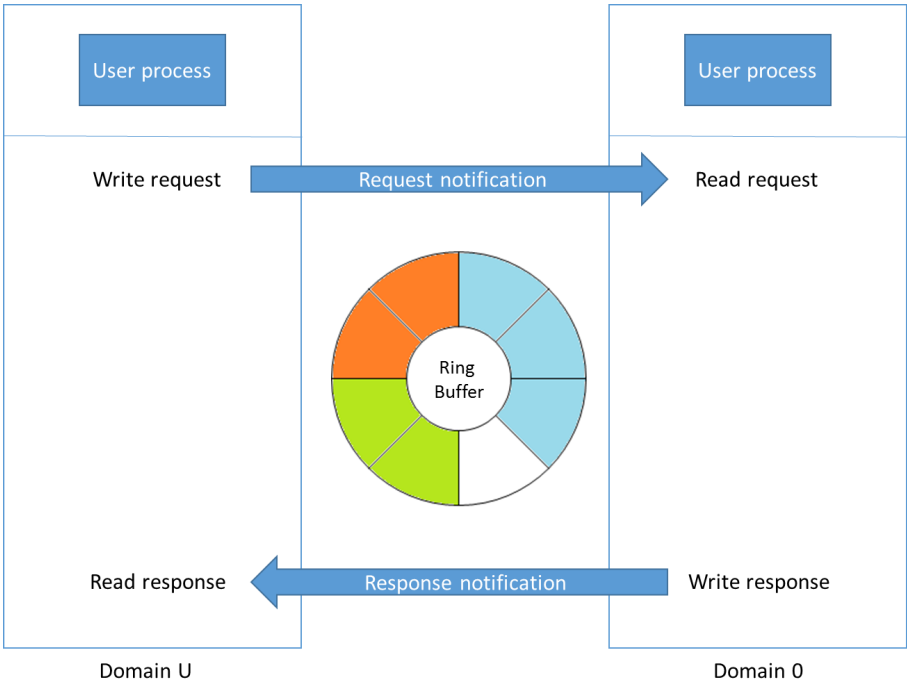



Figure 2.12: Ring I/O buffer

## Shared Pages

### Grant Table

Grant tables are a mechanism provided by the Xen hypervisor for sharing and transferring frames between the domains. It is an interface for granting foreign access to machine frames and sharing memory between underprivileged domains provided by the Xen hypervisor. In Xen, each domain has a respective grant table data structure, which is shared with the Xen hypervisor. The grant table data structure is used by Xen to verify the access permission other domains have on the page allocated by a domain [3].

### Grant References

Grant references are the entries in a grant table. A grant reference entry has every detail about the shared page. The Xen hypervisor virtualizes the physical memory, it is difficult to know the correct machine address of a frame for a domain. The biggest difficulty in sharing the memory between domains is knowing its correct machine address. A grant reference removes the dependency on the real machine address of the shared page. Hence, a grant entry makes it possible to share the memory between domains.[15, 9, 3]

# Chapter 3

# System Introduction

## 3.1 Design Goal

The goal of the IDDR system is to provide full isolation between a device driver and the monolithic kernel and at the same time avoid modifications to the device driver code. The goal of the thesis is to minimize the performance penalty because of the communication between the domains. In the thesis, we explore opportunities to minimize the overhead of the communication module in the IDDR system.

### Performance Improvement

The IDDR system is a re-implementation of the Xen's isolated driver domain. Even though the IDDR system provides better robustness for the operating system, it deteriorates the

performance. The reasons for the performance deterioration is due to data copy overhead and overhead of communication between the domains.

**Copy Overhead:** For data intensive operations such as read and write, the IDDR system transfers data between the hardware and the driver domain. It also transfers the same data between the driver domain and the application domain. The extra copy from the driver domain to the application domain is also one of the reasons for the performance degradation of the system.

**Communication Channel Overhead:** The IDDR system runs a device driver in a separate domain called the driver domain. The application domain and the driver domain communicate with each other in order to send requests and get responses from the device driver. The communication between domains add an overhead to the system performance. Our goal is to minimize the overhead during communication between the driver domain and the application domain.

## 3.2 Isolated Device Driver Properties

This section covers the properties of the base IDDR system. As we are explore the opportunities to improve the performance of the base IDDR system, it is necessary that these properties are not compromised.

### Strong Isolation

One of the main properties of the IDDR system is strong isolation. The IDDR system adds an extra layer of isolation in the design which provides fault isolation between the kernel and the device driver. The strong isolation also adds the ability to manage device drivers independently, thus, increasing the availability of the system during maintenance of a device driver.

### Compatibility and Transparency

The extension of existing OS structures usually results in a large number of broken applications. As a specific example, in the microkernel architecture the functional units of an OS were divided into discrete parts in order to achieve the isolation between the kernel components. As a result, the APIs visible to applications were changed [24], which broke the large number of applications. In order to provide compatibility with applications, many microkernel architectures provided an emulation layer [24] for the OSes. The IDDR system maintains compatibility between existing device drivers and applications.

## 3.3 System Overview

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

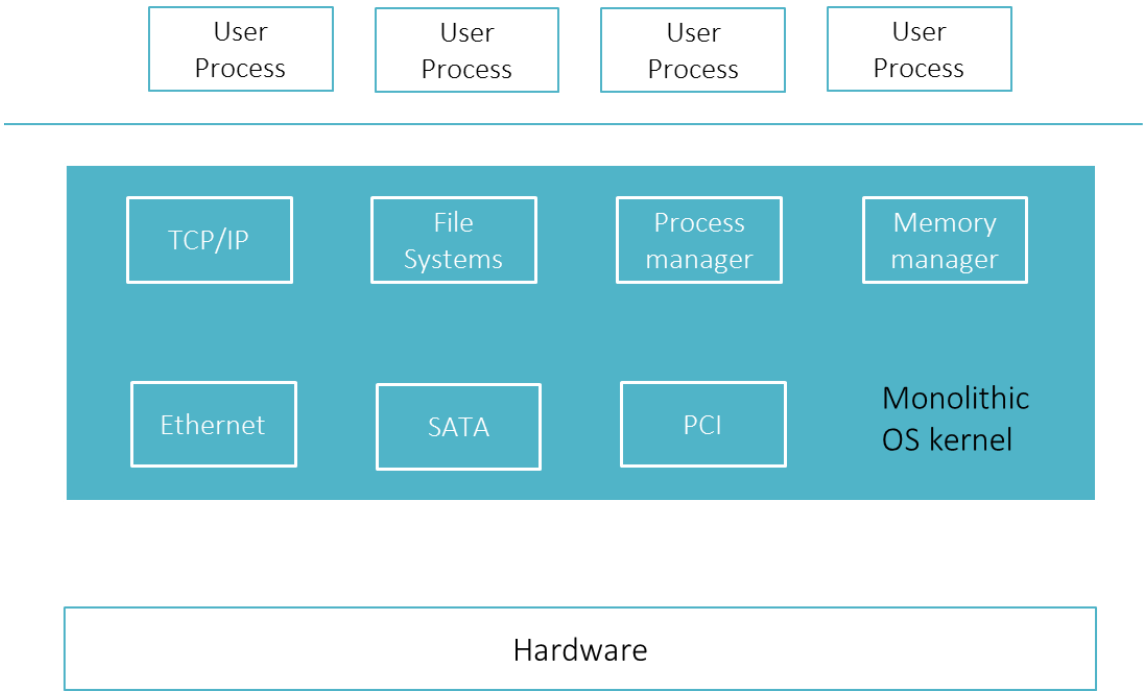



Figure 3.1: Architectural overview of a modern OS

The Figure 3.2 shows that the IDDR system partitions an existing kernel into multiple independent components. The user applications and Linux kernel run in a domain called the *application domain*. The device driver, which needs to be isolated from the 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.

As Section 5.1 describes, the goal of the IDDR system is to provide the isolation between the device driver and the kernel. However, a device driver is dependent on the kernel components

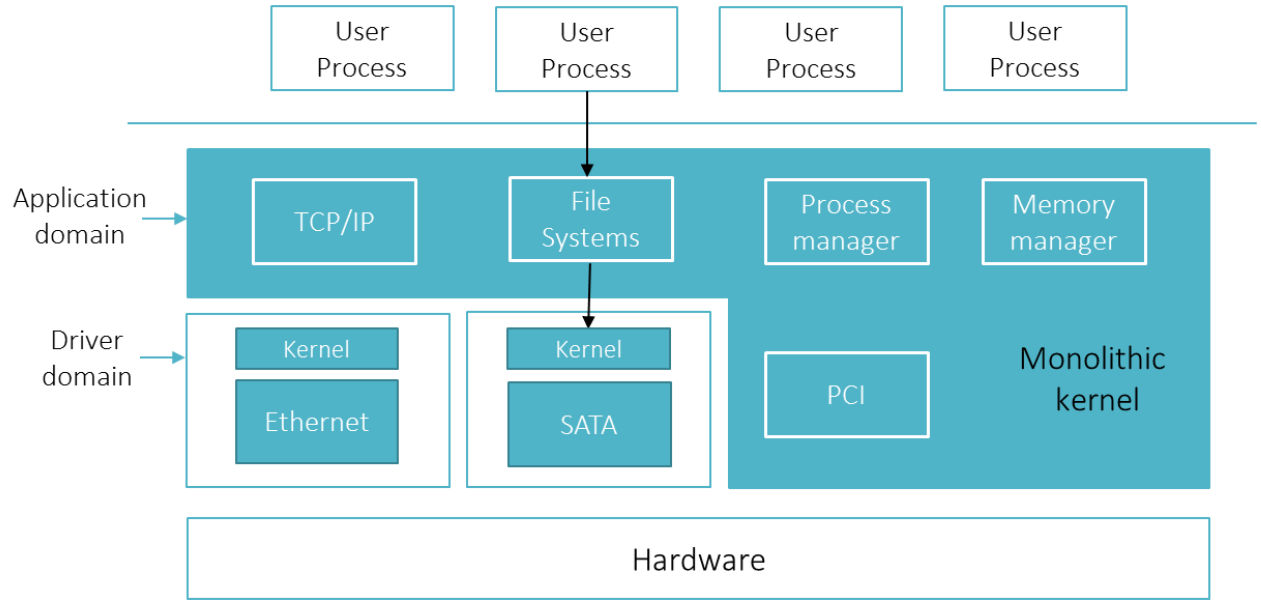



Figure 3.2: Architectural overview of the base IDDR system

such as a scheduler and memory management unit. In order to remove the dependency, the device driver runs closely with another instance of a kernel. Even though the dependency is removed, it is not possible to run multiple kernels over the common hardware without a virtual machine monitor. Thus, a VMM is introduced into the design to run multiple kernels on a common hardware.

## 3.4 System Components

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

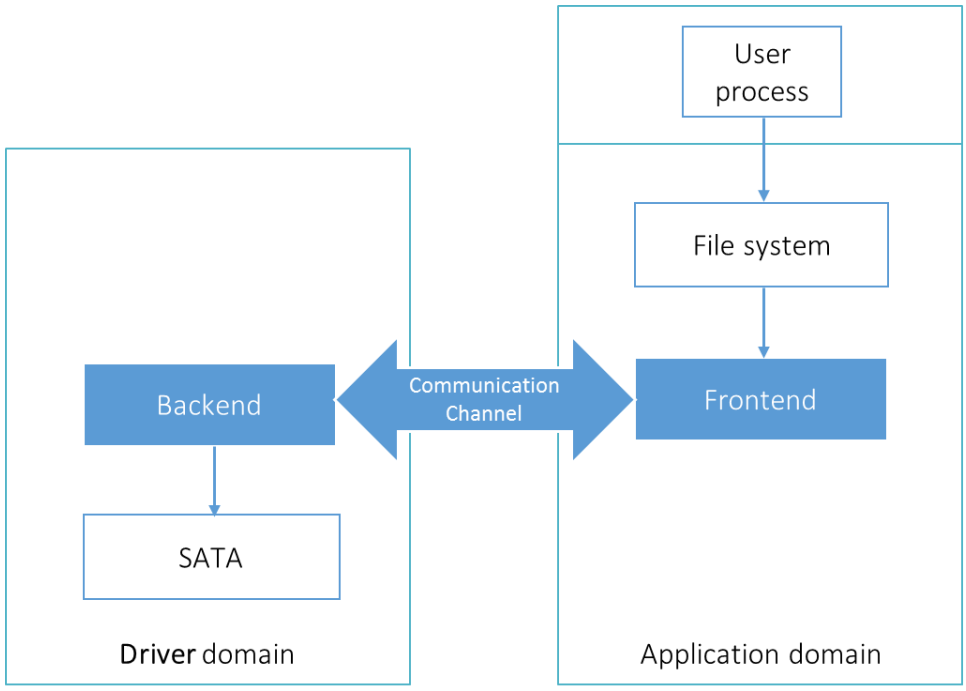



Figure 3.3: System Components

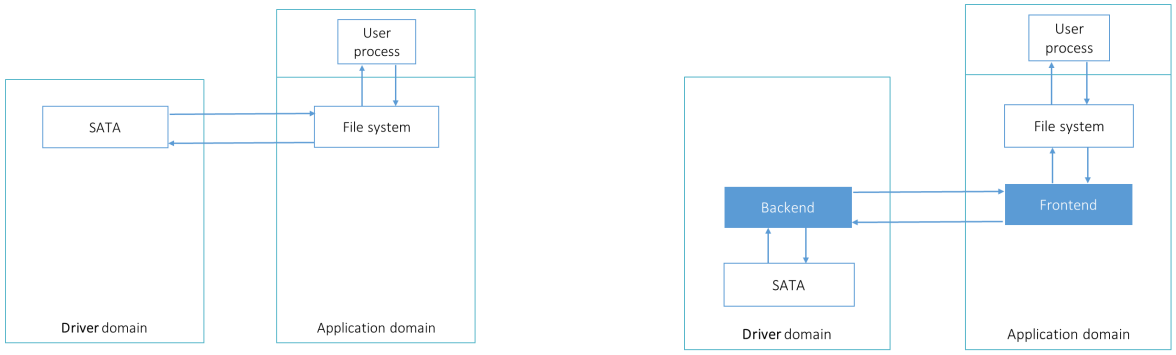
### 3.4.1 Frontend Driver

As mentioned earlier in Section 3.2, transparency and compatibility are the properties of the IDDR system, which requires us to avoid any changes to the kernel, as well as the device driver. In the IDDR system, the device driver runs in the driver domain and user applications run in the application domain. Without making any changes to the kernel or applications, it is not possible for applications to send requests to the driver in the driver domain, as user applications do not know about the isolated device driver. The IDDR system runs a piece of a code called the *frontend driver* in an application domain. The *frontend driver* acts as a substitute for the device driver. The main functionality 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 ends corresponding requests.

### 3.4.2 Backend Driver

In a Linux system, the device driver provides an interface to accept requests from user applications. However, the device driver is not capable of accepting the requests from applications running in a different domain. It cannot send responses back to the application domain without making any changes to the device driver code. In order to avoid making any changes to the device driver or the kernel, a piece of code called the *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 device driver. The *backend driver* sends the responses and notifies the application domain after receiving the responses from the device driver.




(a) Conceptual design of the driver domain

(b) Backend and frontend driver

Figure 3.4: Role of the frontend and the backend driver

### 3.4.3 Communication Module

The communication module is the communication channel between the *frontend driver* and the *backend driver*. Unlike the *backend driver* and the *frontend driver*, the communication module is not a separate physical entity or a kernel module. It exists in the *frontend driver* and the *backend driver*. The communication channel is logically divided into three parts.

1. The responsibility of the first part is to share the requests and responses between the driver domain and the application domain.

2. The responsibility of the second part is to share the data of read/write requests/responses.

3. The responsibility of the third part is to notify the domain upon the occurrence of a particular event.

Figure 3.5 illustrates the role of the communication model.

## 3.5 System Design

The following section describes the design of the base IDDR system and the new IDDR system.

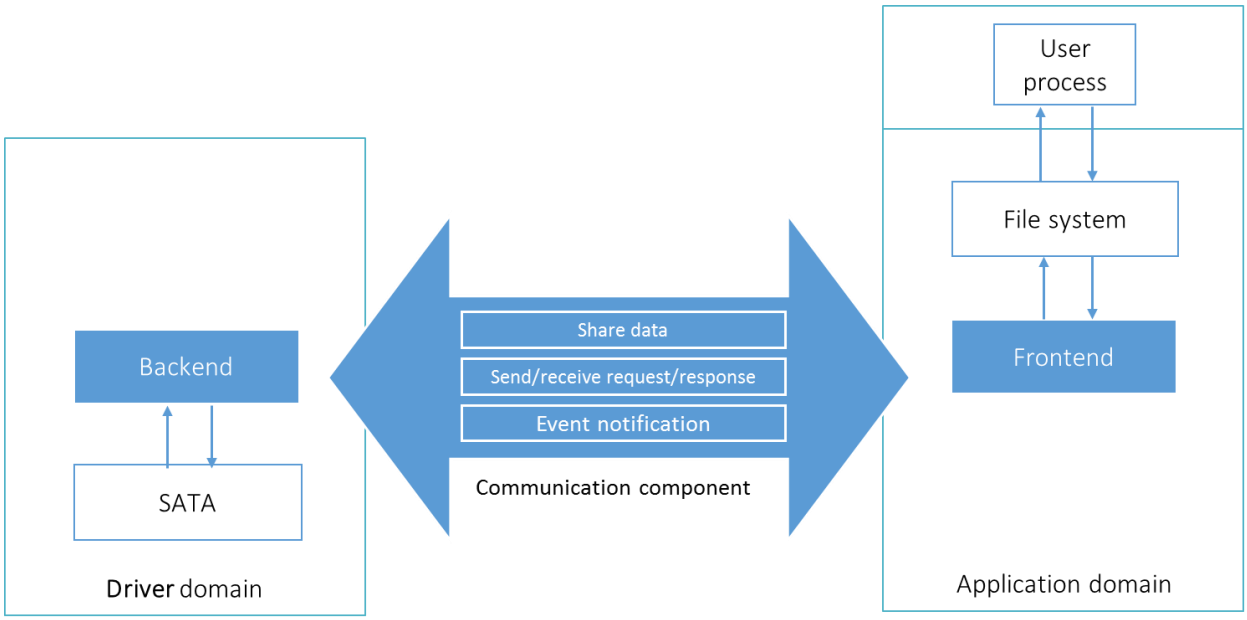



Figure 3.5: Communication Module

### 3.5.1 Communication Module

**Base IDDR System Design:** In the base IDDR system, the frontend driver submits requests to the communication module. The communication module copies data of the write requests in a shared memory. The communication module is responsible for the allocation and de-allocation of the shared memory. Once the sufficient number of requests are 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 a shared request queue.

**Spinning Based IDDR System:** As Section 3.5.1 describes, in the base IDDR system, the communication module notifies the backend driver about the avaiability of reqesuts in the shared queue. A software interrupt is sent to the domain as a notification. Each software interrupt which is sent for the availability of requests, causes the hypervisor to schedule the driver domain. Similarly, a software interrupt, which notifies 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 swtich.

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

### 3.5.2 Frontend Driver

**Base IDDR System Design:** 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 a user application. Similar to the block device drivers, the frontend driver also creates an individualized request queue for each device to accept requests from a user application. If sufficient requests are received, the frontend driver flushes the requests to the communication

channel. The frontend driver receives a software interrupt upon availability of responses in the shared queue. The frontend driver handles the software interrupt by reading data from the shared memory and ending the request in case of a read operation. Otherwise it ends the request without accessing the shared memory.

**New IDDR System Design:** As explained in Section 3.5.1, we introduce an intermediate thread to read responses from the shared queue. The intermediate thread spins for responses. Upon availability of a response, the thread reads the response and ends the corresponding request. If the corresponding request is a read request, then the thread reads the shared data too. However, there still exists an intra-domain context switch between the frontend driver's main thread and the intermediate thread. The main thread is responsible for reading the requests from the frontend driver request queue, and flushing them to the communication channel. The main thread context switches to the intermediate thread, which spins for the responses.

To avoid the intra-domain context switch, we spin the main frontend thread for a short time waiting for responses. If a response is available, then similar to intermediate thread, the main thread reads shared data and the response and ends the corresponding request. However, in case of unavailability of a response, the main thread proceeds to read the new requests from the frontend driver request queue and the intermediate thread checks for the responses.

### 3.5.3 Backend Driver

**Base IDDR System Design:** In the base IDDR system, the backend driver receives a software interrupt from the frontend driver. In the software interrupt handler, the backend driver forwards requests to the device driver. Upon completion of requests, the backend driver reads responses and puts them in the shared queue. It also copies read operation data to the shared memory.

**New IDDR system Design:** As explained in Section 3.5.1, we introduce an intermediate thread to read requests from the shared memory. The intermediate thread spins for requests and upon availability of a request, it forwards the request to the device driver for execution. The backend driver reads the response from the device driver and shares it in the shared queue.

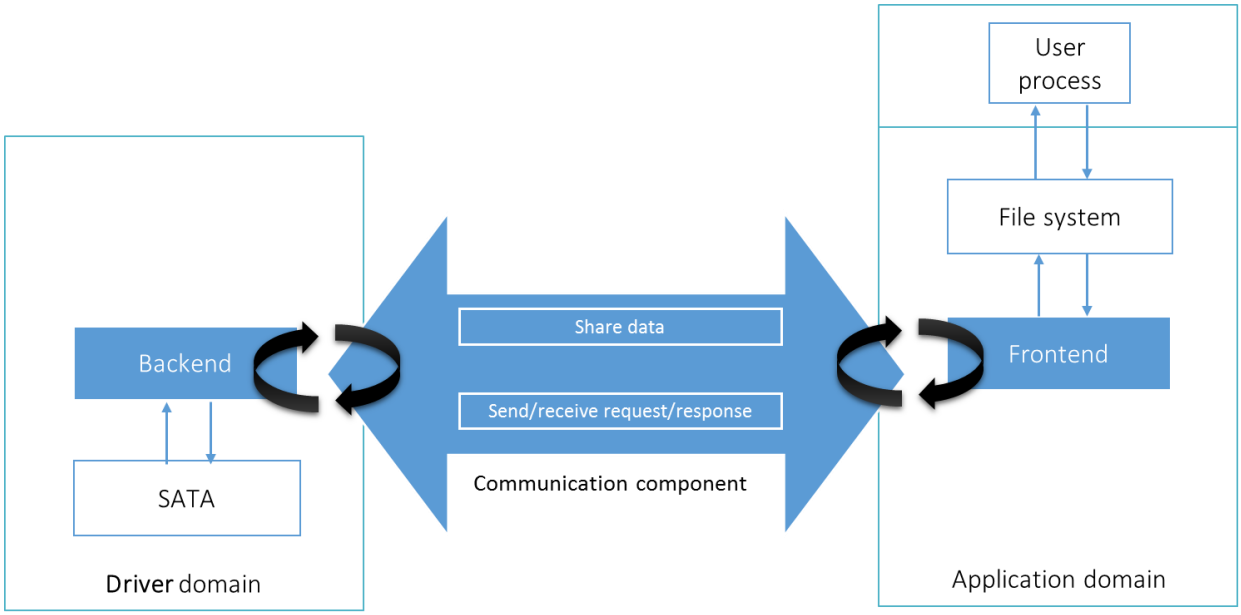



Figure 3.6: Spinning based new IDDR system

# Chapter 4

# Implementation

This chapter describes the specific implementation details of the base IDDR system and the new IDDR system.

## 4.1 Implementation Overview

We implemented the IDDR system with Linux kernel 3.5.0 and Xen hypervisor 4.2.1. In this implementation, we isolate the block device driver from the Linux kernel. The application domain and the driver domain run the same Linux kernel in the IDDR system. The Table 4.1 and Table 4.2 summarize our implementation efforts of the base IDDR system and the new IDDR system respectively.

The IDDR system implementation did not require any changes to the device driver code.

Table 4.1: The base IDDR system implementation efforts.

| Component | Number of Lines |
|---|---|
| Linux Kernel | 6 |
| Xen | 252 |
| Front-end Driver | 611 |
| Back-end Driver | 692 |
| Total | 1561 |

Table 4.2: The new IDDR system implementation efforts.

| Component | Number of Lines |
|---|---|
| Linux Kernel | 6 |
| Xen | 252 |
| Front-end Driver | 712 |
| Back-end Driver | 752 |
| Total | 1722 |

However, we did make a small number of changes to the Xen and Linux kernel to implement a hypercall.

## 4.2 Implementation

Figure 4.1 shows the implementation overview of the new IDDR system.

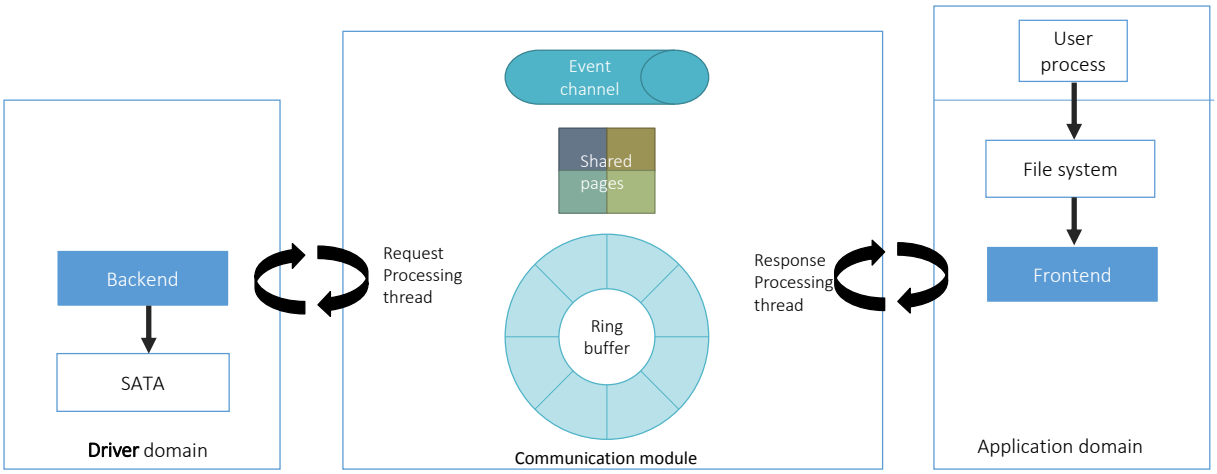



Figure 4.1: Implementation overview of the new IDDR system

### 4.2.1 Communication Module

The most important component in the IDDR system implementation is the communication module. This section describes the implementation details of the communication module of

the base IDDR system and the new IDDR system.

### The base IDDR system

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

1. Share requests and responses between the driver domain and the application domain
2. Share data of 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. The ring buffer is a shared I/O ring explained in section 2.6.2. The part of the ring buffer which shares requests is called the *shared request queue*, and the part of the ring buffer which shares responses is called the *shared response queue*. 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 IDDR system allocates the ring buffer in the initialization stage of the communication module and initializes the ring buffer as the front ring 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 sufficient requests together, the communication module pushes all requests to the shared request queue.

**Shared Memory for Read/Write Data:** The ring buffer cannot hold the data of read responses and write requests. We use the ring buffer only to share requests and responses. In order to share actual data 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.

The frontend driver removes the request from the device driver queue in the application domain, and forwards it to the communication module. The communication module allocates shared memory required to read the data from the backend driver, or to write the data to backend. The communication module then grants access to the driver domain. As the driver domain now has access to the shared memory, the backend driver can access the data from the shared memory.

**Event Notification:** An event channel is a mechanism provided by Xen hypervisor for event notification. The communication module in the base IDDR system uses event channel to send notifications between the domains to notify the availability of requests and responses in the shared request queue and the shared response queue.

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.

**New IDDR System**

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

1. Share requests and responses between the driver domain and the application domain
2. Share the data of read/write requests/responses
3. Wake up the read request thread and the read response thread

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

**Shared Memory for Read/Write Data:** Similar to the base 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 new 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 the new IDDR system we create a kernel thread during an initialization stage of the communication module in the application domain. This new kernel thread is called the *read response thread*. 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[37, 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 the base 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 the new 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 the base 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.

### Base IDDR System

The core responsibility of the frontend driver in the base 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.

**New IDDR System**

The core responsibility of the frontend driver in the new 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
3. Spinning the main thread
4. End request

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

**Dequeue and Submit Request:** Similar to the base 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.

**Spinning The Main Thread:** In the new IDDR system, to avoid the intra-domain context switch between the read response thread and the main frontend driver thread, the main frontend driver thread spins for a short time and checks the availability of a response. If the response is available then the frontend driver main thread reads the response and proceeds to end the request. Otherwise it continues to remove the request from the device driver queue.

**End Request:** Similar to the base IDDR system, we maintain a shadow table of all requests received in the frontend driver request queue. The shadow table is used for ending the corresponding request of the read response.

### 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 re-

ceiving 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 base IDDR system:**

   The goal of this thesis is to explore the performance improvement opportunity in the Xen's isolated driver domain and implement it. However, the source code for 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 |

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 base Isolated Device Driver (IDDR) system. We implemented the spinning based communication channel over the base IDDR system. We call it as the new IDDR system.

In order to prove the performance improvement of the new IDDR system over the base 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 base 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 base 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 base IDDR system with the new IDDR system. The Comparison shows that the new IDDR system performs better than the base 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 Base IDDR System

As per our first goal of evaluation mentioned in Section 5.1, we compare the base 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.

### Base 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 base 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.

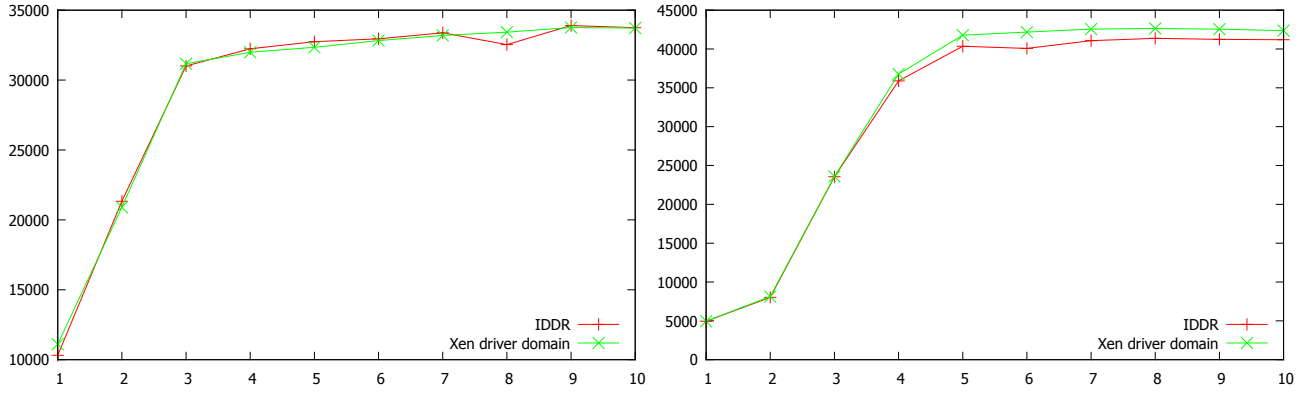
We insert a ramdisk and the base IDDR system's backend module in the domain 0 and the frontend module in the guest domain (domain U). We format and mount the ramdisk with ext2 file system and run the SysBench benchmark on it.

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 base 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 base IDDR system differs by 3%-4% when compared to xen split device driver. On a ramdisk, the throughput of the base 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: Base IDDR system vs Xen split driver

## 5.4 Base IDDR System vs New IDDR System

We measure and compare the performance of the base IDDR system with the new 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 base IDDR and the new 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 new 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 new IDDR system performs better than the base IDDR system, but as number of SysBench threads increases, the throughput

of the new IDDR system decrease.

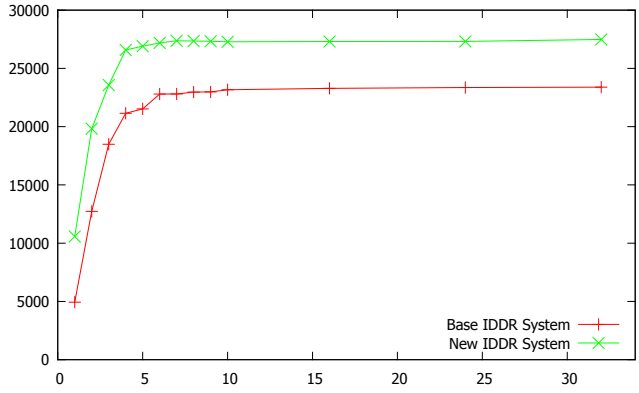
Similarly, the Figure 5.2c and Figure 5.3c compares the mix random read and write performance. In case of mixed reads and writes, most of the time is spent in writing the data. The throughput of the system is dominated by the write performance of it. Figure 5.2c and Figure 5.2b show that curves are similar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

**Observation :** The performance analysis of the base 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.

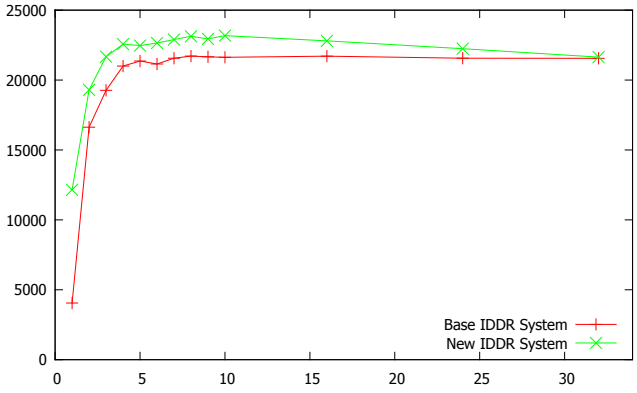
It is not the case with the new IDDR system. Since both the application domain and driver domain spins for the request and responses, when a data workload is low, we observe a more performance gain. When data workload is low, the CPU is idle. Since the frontend and backend driver spin for the request 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 new IDDR system decreases

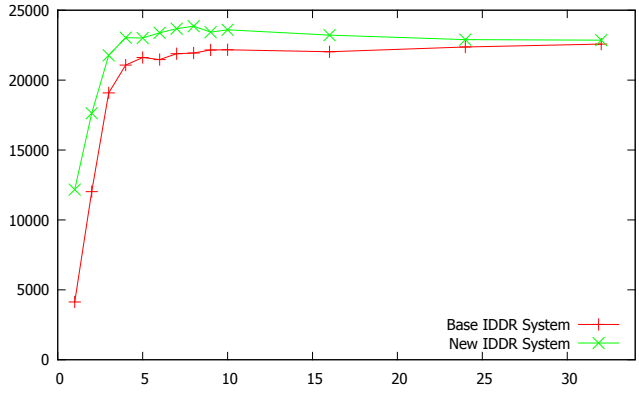
and matches that of the base IDDR system. Heavy workload denotes the more number of SysBench thread, which denotes the high CPU utilization. Since our system exploits the idle CPU to get the better performance, when the CPU is already under heavy load, the performance of the system decreases.




(a) Random reads

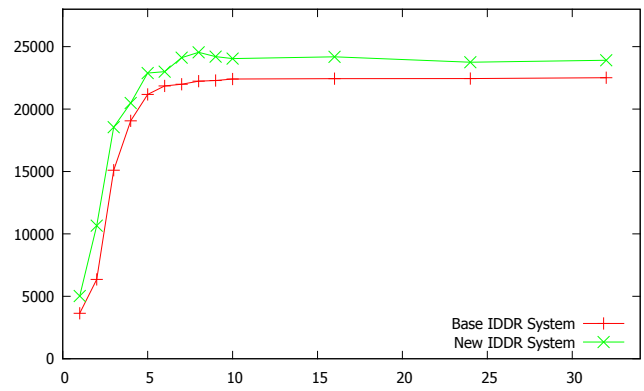



(b) Random writes

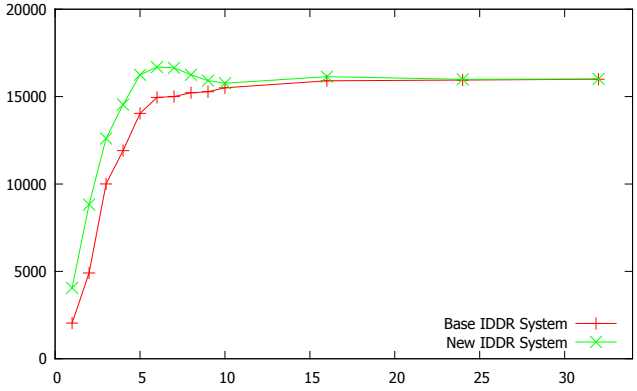



(c) Random reads writes

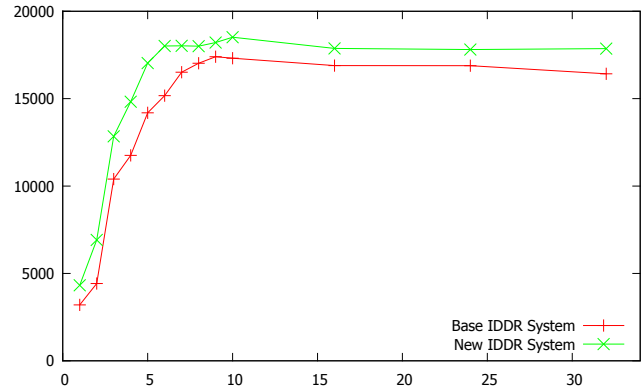
Figure 5.2: Random reads and writes on a Ramdisk




(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 briefly discusses work closely related to the IDDR System. Our work is divided into two parts:

1. Implementation of the isolated driver domain which improves the reliability of the system

2. Performance improvement of inter-domain communication in the isolated driver domain

This chapter is divided into two sections. Section 6.1 discusses work on improving the reliability of a system and Section 6.2 discusses work which concentrates on improving inter-domain communication.

## 6.1 Reliability of the System

The goal of the baseline implementation of the IDDR system is to increase the reliability of an operating system. In this section we describe research in the operating systems area which focuses on increasing the reliability of a system.

### 6.1.1 Driver Protection Approaches

A faulty device driver causes a significant number of failures in the Linux kernel [41, 1]. In the past, research in areas, such as hardware based driver isolation, language based driver isolation and user level device drivers, has contributed towards isolating device drivers, hence increasing the reliability of a system.

Numerous implementations run device drivers in the user mode. Even though user mode device drivers allow user level programming and a good fault isolation between components, they suffer from poor performance [8] and also require re-writing of the existing device drivers. The user mode device drivers also lack compatibility [28]. Microdrivers [20] extend the user mode device driver research and split a device driver into two parts. In Microdrivers, performance critical operations of the device driver run in the kernel and the rest of the driver code runs in a user mode process. Microdrivers deliver good performance and compatibility.

Apart from user mode device driver, some approaches use hardware based driver isolation

to achieve the isolation between components. Nooks is one example of such approaches. Nooks [40] maintains the monolithic kernel structure, and focuses on making device drivers less vulnerable. It creates a lightweight protection domain around each device driver. The domain is created by wrapping a layer of protective software around a device driver. The wrapper layer monitors all interactions between the driver and the kernel, and protects the kernel from a faulty device driver. Nooks requires device drivers to be modified as their interaction with the kernel changes.

SUD [12] runs unmodified Linux device drivers in the user space. It uses the emulated IOMMU hardware to run a device driver in the user space. Running device drivers in a user space safely isolates a system from a malicious device driver.

Dune [10] is a system that provides an application direct and safe access to the hardware features, such as page tables, tagged TLBs and ring protection. It uses virtualized hardware to isolate applications from each other. Dune delivers hardware interrupts directly to applications in order to improve the signal delivery performance. However, Dune does not isolate device drivers from each other.

**Virtualization Based Approaches**

The IDDR system isolates a device driver from a Linux kernel using Xen VMM. Research work, which uses virtualization techniques to isolate the kernel components is related more closely to our work.

Xen isolated driver domain [19] is a device driver isolation architecture presented by Xen. The isolated device driver allows unmodified device drivers to be run in a separate domain and shared across operating system instances. Isolated driver domain protects individual operating systems, from driver failure. The base IDDR system is the re-implementation of Xen isolated driver domain.

LeVesseur et. al. [29] presents a virtualization based system to reuse unmodified device drivers. It also improves system reliability. In this approach, an unmodified device driver is run with a kernel in a separate virtual machine. The main goal of this system is to reuse the device driver across different operating systems. However, the system isolates faults caused by a device driver by running a device driver in separate virtual machine.

VirtuOS [35] is a library level solution that allows processes to directly communicate with a domain. 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 Other Approaches

#### Kernel Based Approach

In this approach, systems provide a better fault containment by disintegrating the kernel functionality as system components. Microkernels are examples of such approaches. They provide fault containment by providing an isolation between the system components. In the

microkernels such as Mach [6] and L4 [31], only essential functionalities like memory management, interprocess communication, scheduling and low level device drivers are implemented in the kernel. Remaining system components, like file system and process management are implemented as user processes.

Some of the recent work also uses the similarity in the abstraction provided by the hypervisor and microkernel. Microvisor [25] is a kernel that satisfies the combined objectives of microkernels and hypervisors, and provides an abstraction of a virtual machine, where a guest OS schedules activities on one or more VCPUs.

## 6.2 Inter-domain Communication

In the past numerous work on inter domain communication mechanisms was presented. In Xen VMM, a domain communicates with the privileged domain through the split device driver mechanism [19]. In a way, the split device driver mechanism is a restricted inter domain communication path. The Xen split driver faces the performance issues because of the overhead of numerous context switches in form of virtual interrupts. It also incurs an overhead due to extra data copy and page flipping [42].

In order to overcome the page flipping performance overhead, Xen hypervisor also provides a UNIX domain socket like interface called XenSocket [42]. XenSocket provides a high throughput inter-domain communication. It replaces the page flipping design of the split driver. However, XenSocket needs an existing socket interface APIs to be changed.

Fido [13] is a shared memory based inter domain communication mechanism. Fido implements the fast inter-domain communication mechanism by reducing data copies in the Xen hypervisor. In contrast, the IDDR system improves the inter domain communication mechanism of Split device drivers by avoiding the context switches. Also, Fido removes overhead with zero copy by sacrificing the security and protections 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] 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.

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

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.