Performance analysis of driver domain.

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

Dec 12, 2013

Blacksburg, Virginia

Keywords: Device driver, Virtual machines, domain,

Device driver isolation using virtual machines.

Sushrut Shirole

(ABSTRACT)

In majority of today's operating system architectures, kernel is tightly coupled with the device drivers. In such cases, failure in critical components can lead to system failure. A malicious or faulty device driver can make the system unstable, thereby reducing the robustness. Unlike user processes, a simple restart of the device driver is not possible. In such circumstances a complete system reboot is necessary for complete recovery. In a virtualized environment or infrastructure where multiple operating systems execute over a common hardware platform, cannot afford to reboot the entire hardware due to a malfunctioning of a third party device driver.

The solution we implement exploits the virtualization to isolate the device drivers from the kernel. In this implementation, a device driver serves the user process by running in a separate virtual machine and hence is isolated from kernel. This proposed solution increases the robustness of the system, benefiting all critical systems.

To support the proposed solution, we implemented a prototype based on linux kernel and Xen hypervisor. In this prototype we create an independent device driver domain for Block device driver. Our prototype demonstrate that a block device driver can be run in a separate

domain.

We isolate device drivers from the kernel with two different approaches and compare both the results. In first approach, we implement the device driver isolation using an interrupt-based inter-domain signaling facility provided by xen hypervisor called event channels. In second approach, we implement the solution, using spinning threads. In second approach user application puts the request in request queue asynchronously and independent driver domain spins over the request queue to check if a new request is available. Event channel is an interrupt-based inter-domain mechanism and it involves immediate context switch, however, spinning doesn't involve immediate context switch and hence can give different results than event channel mechanism.

Acknowledgments

Acknowledgments goes here

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A system is judged by the quality of the services it offers and 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 characteristics of the operating systems which make them unstable are size and complexity.

Coverity's analysis of the Linux kernel code found 1000 bugs in the source code of version 2.4.1 of the Linux kernel, and 950 bugs still in 2.6.9. This study also shows that 53% of the bugs are present in the device driver portion of the kernel [2].

In order to protect against bugs, operating systems provide protection mechanisms. These protection mechanisms protect resources such as memory and CPU. Memory protection is the protection mechanism provided in a Linux kernel. Memory protection is a way to control 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 [19, 39]. However, kernel modules do not have the same level of protection the 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 the memory of 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

In the past, virtualization based solutions, which were intended to increase the reliability of a system, were proposed by LeVasseur et. al. [29] and Fraser et. al. [21]. Frazer et. al. proposed the Xen isolated driver domain. In a virtualized environment, virtual machines are unaware of and isolated from the other virtual machines. Malfunctioning of one virtual machine cannot spread to the others. Hence, use of virtual machines provide extremely good 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 and the kernel in separate virtual machines. Xen hypervisor provides a platform to the isolate device driver from the monolithic kernel called the isolated driver domain based on a split device driver model [1].

The Xen virtual machine monitor (VMM) does not include device drivers for all the devices, adding support for all the devices would be a duplication of effort. Instead, Xen delegates hardware support to guests. The guest typically runs in a privileged domain, although it is possible to delegate hardware to guests in other domains. The model in which hardware support is delegated to a guest is called a split device driver model [14]. Xen uses the same split device driver model in the isolated driver domain as shown in the figure 1.1. Xen has a frontend driver in the guest domain and a backend driver in the driver domain. The frontend and the backend driver transfer data between domains over a channel that is provided by

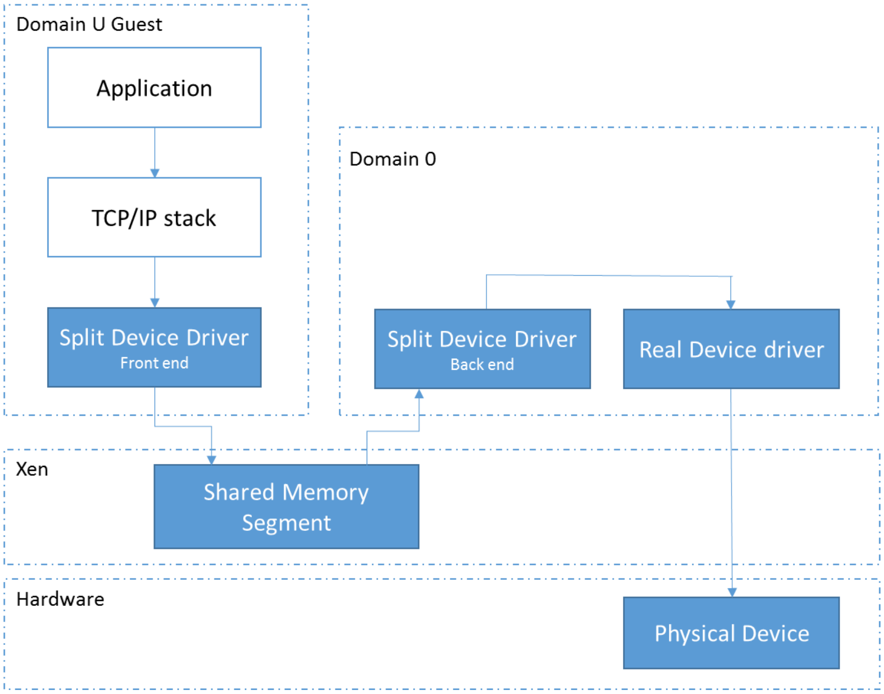


Figure 1.1: Split device driver model

the Xen VMM. Within the driver domain, the backend driver is used to de-multiplex incoming data to the device and to multiplex outgoing data between the device and the guest domain [1].

The Xen's isolated driver domain follows the split device driver architecture of Xen VMM. In the isolated driver domain, 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 the advances in virtualization technology, the overhead of communication between guest domain and driver domain significantly affects the performance of applications [7, 41, 32]. The Xen's isolated driver domain follows an interrupt based approach in the communication channel [7]. In this communication model, frontend and backend notifiy 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 might need a context switch [7]. The context switch can cause an unavoidable system overhead [30, 33]. The interrupt based notification system might cause an overhead in the communication channel of the isolated driver domain.

1.2 Proposed Solution

In this thesis, we propose and evaluate an optimization for improving the performance of the communication between guest domain and driver domain. We propose a solution in which a thread in the backend driver spins for the service requests, and the frontend driver spins for the availability of the corresponding responses. Multiprocessors have better concurrency than single core processors. On multi core processors, spinlocks have better performance than uniprocessor system. On the other hand, a context switch takes a significant amount of time, so it is more efficient for each process to simply spin while waiting for a resource in a multiprocessor environment. Since our solution follows a spinlock based approach, it performs better than the approach used in the original implementation.

The source code for isolated driver domain is not available in the open source Xen hypervisor code. As explained in the Section 1.1, isolated driver domain uses Xen's split device driver approach [21]. In this thesis, we re-implement the Xen's isolated driver domain, we refer to our implementation as Isolated Device Driver (IDDR). Our solution to improve the performance of the Xen isolated driver domain is implemented over the IDDR base code.

The performance of the system is evaluated for block devices such as ramdisk device, loop device, and SATA disk. The block device is formatted with a file system and the IDDR system is evaluated by measuring the performance of the system with fileIO/SysBench benchmark. The integrity of the system is checked by executing dd read/write, with and without read ahead, file system tests on the variety of block devices. The evaluation of our solution shows that the performance of the system can be improved by avoiding the context switches in the communication channel.

1.3 Core Contributions

The core contributions of this project are listed below:

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

1.4 Organization

This section gives the organization and roadmap of the thesis.

1. Chapter 2 gives the background on Processes, Threads, Memory Protection, Virtualization, Hypervisor and Inter-domain Communication.
2. Chapter 3 gives the introduction to design of the system to isolate device driver.
3. Chapter 4 discusses the detailed design and implementation to isolate the device driver.
4. Chapter 5 evaluates the performance of the Independent device driver with different designs.
5. Chapter 6 reviews the related work in the area of kernel fault tolerance.

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

Chapter 2

Background

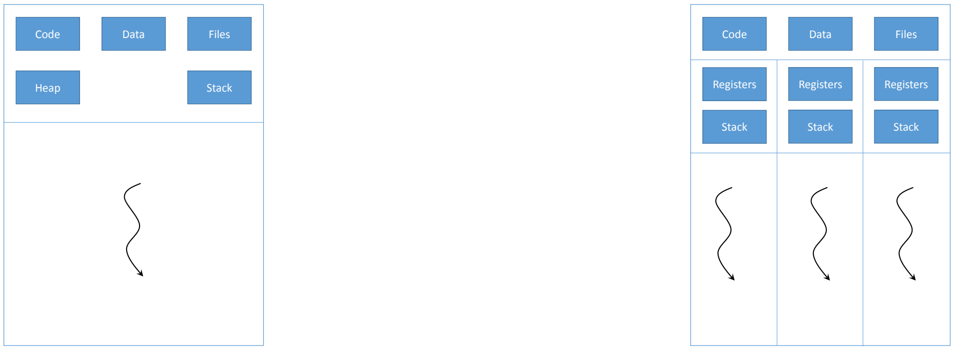
This section gives a background on operating system concepts such as Processes, threads, Memory protection, Virtualization and Hypervisor.

2.1 Processes and threads

Process: Process is a program in execution or an abstraction of a running program. Process can be called as the most central concept in an operating system [39].

Threads: Each process has an address space. A process has either single or multiple threads of control in that same address space. Threads run as if they were separate processes although they share the address space [39].

Thread is also called as a light-weight process. The implementation of threads and processes differ in each operating system, but in most cases, thread is contained inside a process. Multiple threads can exist within the same process and share resources such as code, and data segment, while different processes do not share these resources. If a process has multiple threads then it can perform more than one task at a time.



(a) Single threaded process

(b) Multithreaded process

Figure 2.1: Thread

2.1.1 Context Switch

Multithreading is implemented by time division multiplexing on a single processor. In time division multiplexing, the processor switches between different threads. The switch between threads is called as the context switch. The context switch makes the user feel that the threads or tasks are running concurrently. However, on a multi-processor system, threads can run truly concurrently, with every processor or core executing a separate thread simultaneously.

In a context switch the state of a process is stored and restored, so that the execution can be resumed from the same point at a later time. The state of the process is also called as a context. The context is determined by the processor and the operating system. Context switching makes it possible for multiple processes or threads to share a single processor. Usually context switches are computationally intensive. Switching between two process requires good amount of computation and time, to save and load registers, memory maps, and updating various tables [39] .

2.1.2 Spinlocks

Spinlock is one of the locking mechanisms designed to work in a multi-processing environment. Spinlock causes a thread trying to acquire the lock to spin in case of unavailability of it. Spinlocks are similar to the semaphores, except that when a process finds the lock closed by another process, it spins around continuously till it obtains the lock. Spinning is implemented by executing an instruction in a loop [10].

In a uniprocessor environment, the waiting process keeps spinning for the lock. However, the other process holding the lock might not have a chance to release it, because of which spin lock could deteriorate the performance in a uniprocessor environment. In a multi-processor environment, spinlocks can be more efficient. On the other hand, 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 the resource [10]. As a spinlock avoid overhead of re-scheduling and context switching, it can be more efficient if threads are likely to be blocked for a short period. For same reason, spinlocks are often used inside operating system kernels.

Adaptive Spinning is a spinlock optimization technique. In the adaptive spinning technique, the duration of the spin is determined by policy decisions based on the rate of success and failure of recent spin attempts on the same lock. Adaptive spinning helps thread to avoid spinning in futile conditions.

2.2 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 the hardware functions. Device drivers are hardware dependent and operating system specific. When a program invokes a routine in the driver, the 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 recieving the data. The Linux distinguishes between three fundamental device types: Character device, Block device and Network interface. Each kernel module usually implements one of these types.

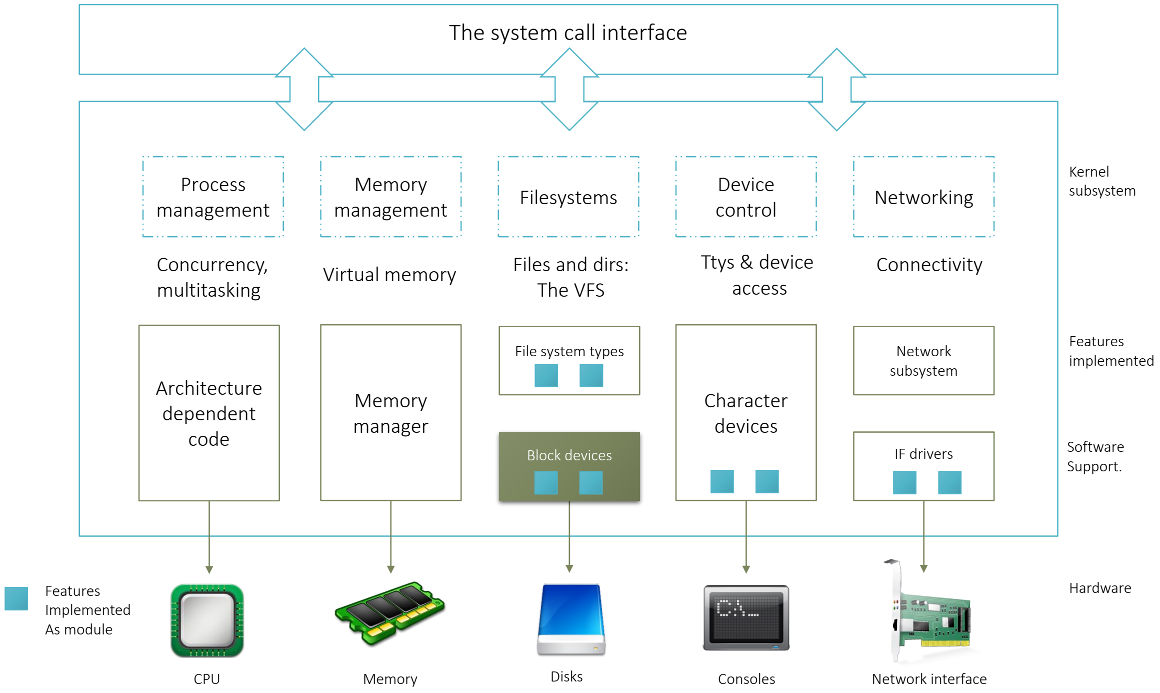


Figure 2.2: Split view of a kernel

2.2.1 Block device driver

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

Network interfaces: 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: A block device is a device that can host a filesystem. Block devices are accessed by filesystem nodes in the /dev directory. In most unix implementation, 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 a block device the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus block drivers have a completely different interface to the kernel than char drivers. Examples of a block devices are disks and CDs [16].

Request processing in a block device driver

The core of every block driver is its request function. The request function is called whenever the driver needs to process reads, writes, or other operations on the device. The request function does not need to actually complete all of the requests on the queue before it returns. However, it ensures that all the requests are eventually processed by the driver. The block device driver request method accepts a parameter, a pointer to the request queue. Every device has a separate request queue. A spinlock is needed as part of the queue creation process. A request function of the device is associated with the request queue in its creation process. The kernel hold the spinlock, whenever the request function is called. As a result, the request function runs in an atomic context.

Each request structure represents one block I/O request, however, it might be formed through a merger of several independent requests at a higher level. The kernel may join multiple requests that involve adjacent sectors on the disk, but it never combines read and write operations within a single request structure. A request structure is implemented as a linked list of bio structures. The bio structure is a low-level description of a portion of a block I/O request [16].

The bio structure The kernel describes a read or write operation of a block device in the form of a bio structure. The bio structure has a everything that a block device driver needs to execute the request, without referencing to the user space process that caused the request to be initiated. After creation, the bio structure is handed to the block I/O code. The block I/O code merges the bio structure into an existing request structure otherwise it creates a new request [16].

2.3 Memory protection

The memory protection mechanism of computer systems control the access to objects. The main goal of memory protection is to prevent malicious misuse of the system by users or programs. Memory protection also ensures that each shared resource is used only in accordance with the system policies. In addition, it also helps to ensure that errant programs cause minimal damage. However, memory protection systems only provide the mechanisms for enforcing policies and ensuring reliable systems. It is up to the administrators, programmers and users to implement those mechanisms [39, 24]. Subsection 2.3.1 and subsection 2.3.2

explain the policies implemented at kernel level and user level.

2.3.1 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 kernel. This upper `VM-X Gb` is restricted to `CPL 0` only. The kernel puts its private data structures in the upper memory and always accesses them at the same virtual address. At 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`> entry. If a process tries to access virtual address which is out of `base + bound` address, then error is reported by the operating system, otherwise physical address `base + virtual address` is returned. This allows multiple processes to be in memory with protection. Since address translation provides protection, a process

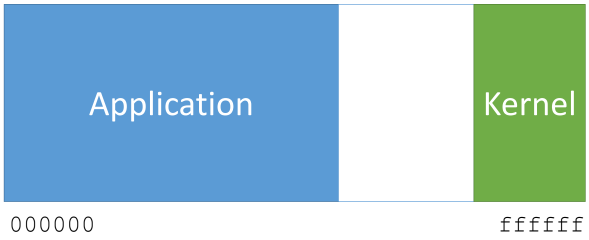
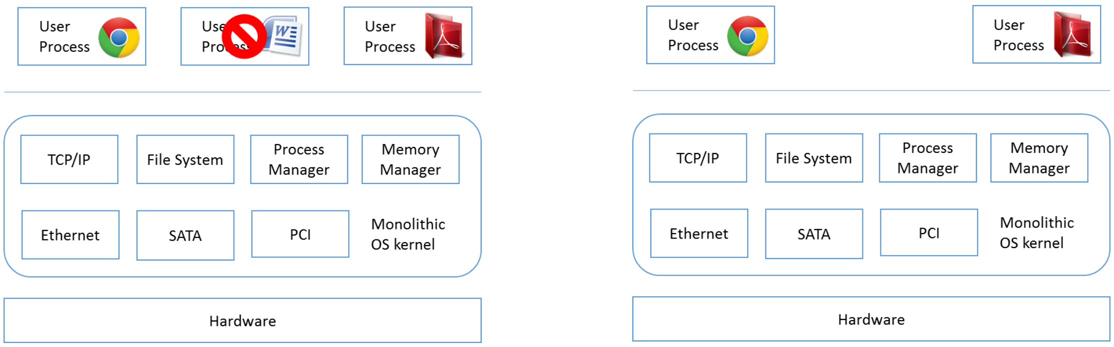


Figure 2.3: Physical memory

cannot access to the address 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 process hits a bug and tries to corrupt the memory out of 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

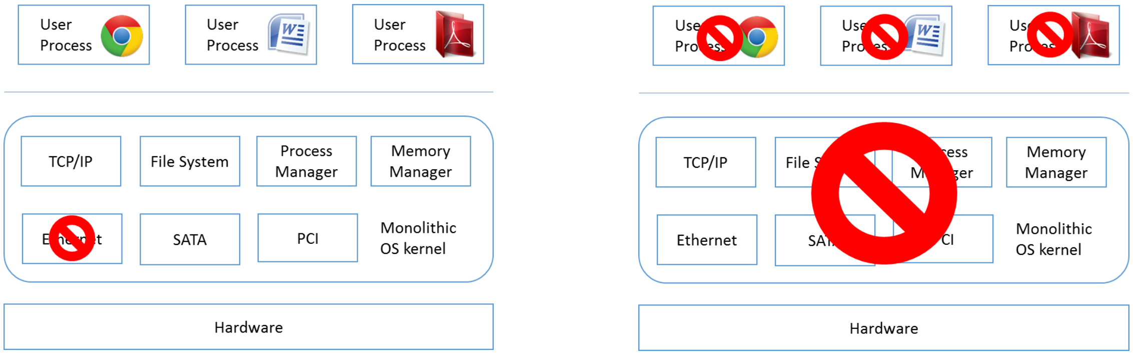
2.3.2 kernel level

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 translation of address from physical to virtual when 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 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, and 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 kernel space.

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



(a) A kernel component hits a bug

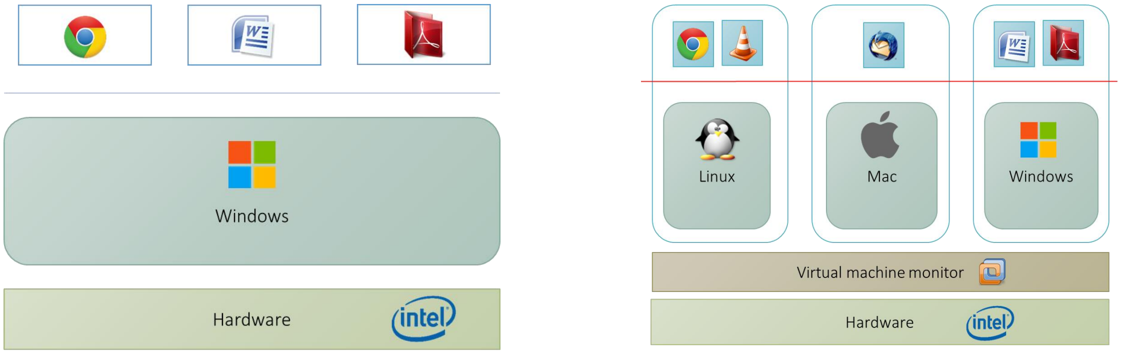
(b) Results in a system crash

Figure 2.5: Kernel level memory protection

2.4 Virtualization

Virtualization, is the act of creating a virtual version of hardware platform, storage device, or computer network resources 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 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 system might require some changes in order to run in a virtualized environment [20]. It has been shown that virtualization can be utilized to provide better security and robustness for operating systems [21, 29, 37].



(a) Operating System Architecture

(b) Virtualization

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

2.4.1 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 software layer specifically responsible for virtualizing a given architecture, and 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 [6].

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 may share the virtualized hardware resources. Among widely known hypervisors are Xen [8, 14], KVM [25, 28], VMware ESX [6],and VirtualBox [13].

There are two types of hypervisors [23]

Type 1 hypervisors are also called as native hypervisors or bare metal hypervisors. Type 1 hypervisor runs 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 hypervisor represents the classic implementation of virtualmachine architectures such as SIMMON, and CP/CMS. Modern equivalents include Oracle VM Server for SPARC, Oracle VM Server, the Xen hypervisor [8], VMware ESX/ESXi [6] and Microsoft Hyper-V.

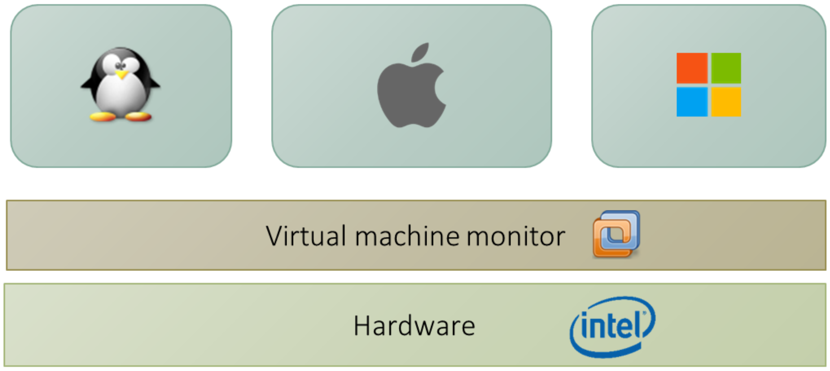


Figure 2.7: Type 1 hypervisor

Type 2 hypervisors are also called as hosted hypervisors. Type 2 hypervisor runs within a conventional operating-system environment. Type 2 hypervisor runs 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 [41, 13].

2.4.2 Xen Hypervisor

Xen [8] is a widely known Type 1 hypervisor that allows execution of virtual machines in guest domains [27]. Figure 2.8 represents a diagram showing the different layers of a

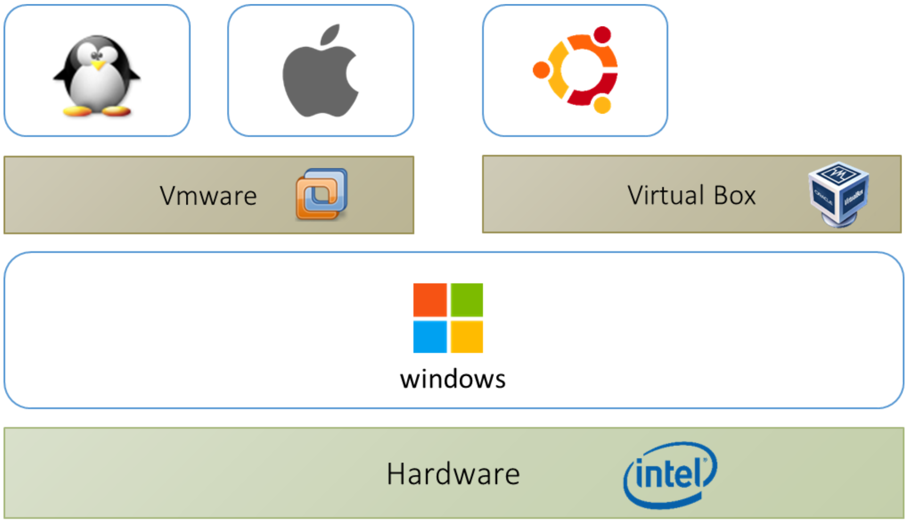


Figure 2.8: Type 2 hypervisor

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 guests to be run. Xen runs guests 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 as domain U. Xen hypervisor does not include device drivers. Device management is included in privileged domain `domain 0`. `Domain 0` uses the device drivers which are present in its guest kernel implementation. However, `domain U` accesses devices using a split device driver architecture.

In the split device architecture a front end 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 a 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, sends it way down to the real device driver. The data is written on 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 [14]. 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 [8, 34, 35].

Hypercalls and events

Hypercalls and event channels are the two mechanisms that exist for interactions between Xen and domains. A hypercall is a software trap from a domain to the Xen, just as a syscall is a software trap from an application to the kernel [3]. Domains use the hypercalls to request

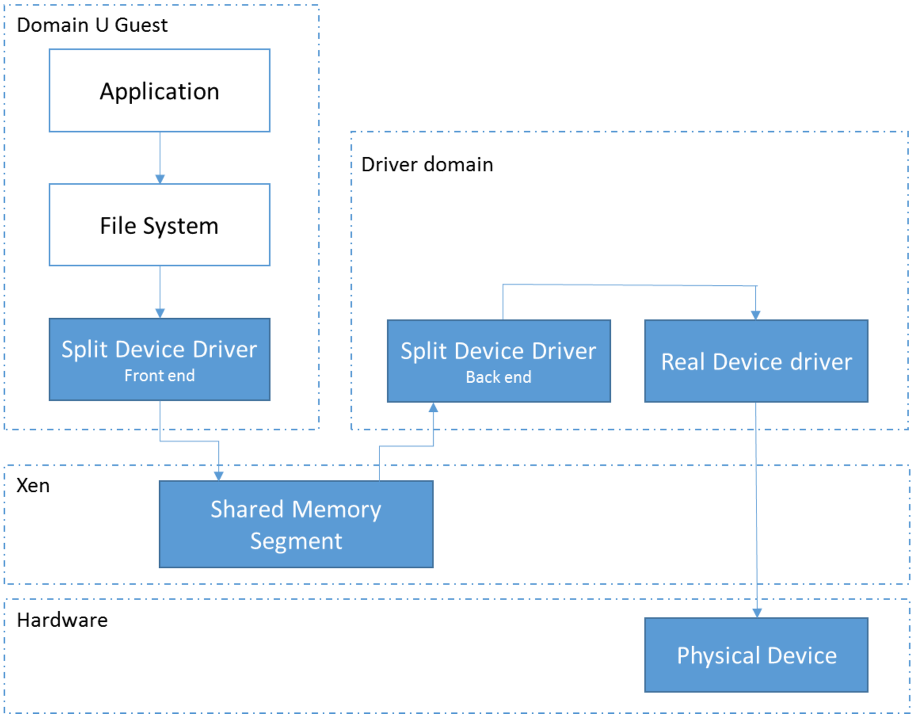


Figure 2.9: Xen split device driver

privileged operations like updating pagetables.

Event channel is to the Xen hypervisor as hardware interrupt is to the operating system. Event channel is used for sending asynchronous notifications between domains. Event notifications are implemented by updating a bitmap. After scheduling pending events from an event queue, the event callback handler is called to take appropriate action. The callback handler is responsible for resetting the bitmap of pending events, and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen readable software flag: this is analogous to disabling interrupts on a real processor. Event notifications can be compared to traditional UNIX signals acting to flag a particular type of occurrence. For example, events are used to indicate that new data has

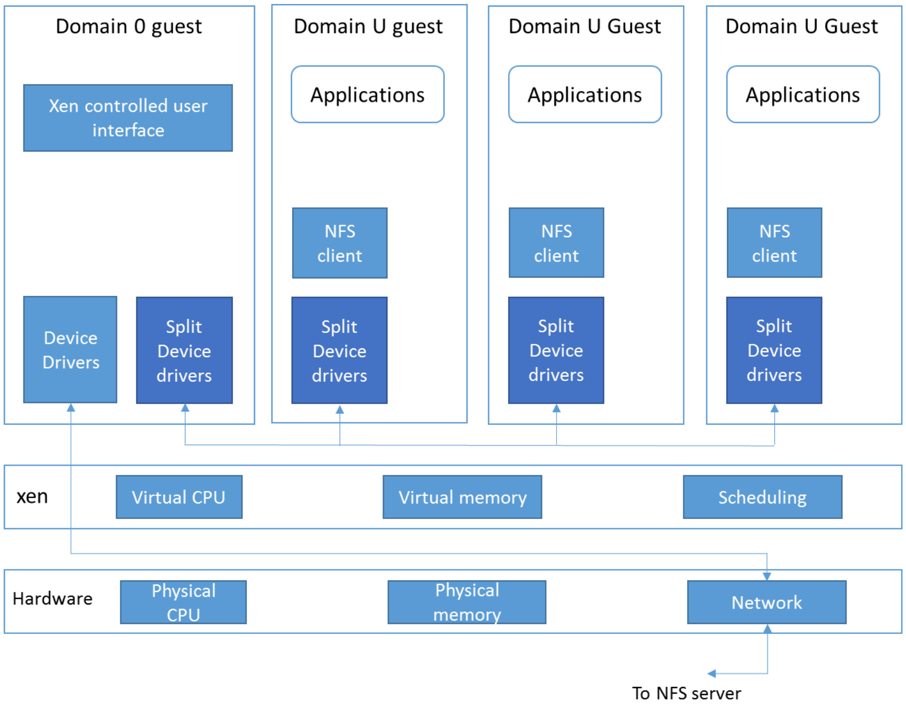


Figure 2.10: Xen

been received over the network, or used to notify 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 2.12 shows the structure of I/O descriptor ring. 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 I/O ring is based around two pairs of producer-consumer pointers.

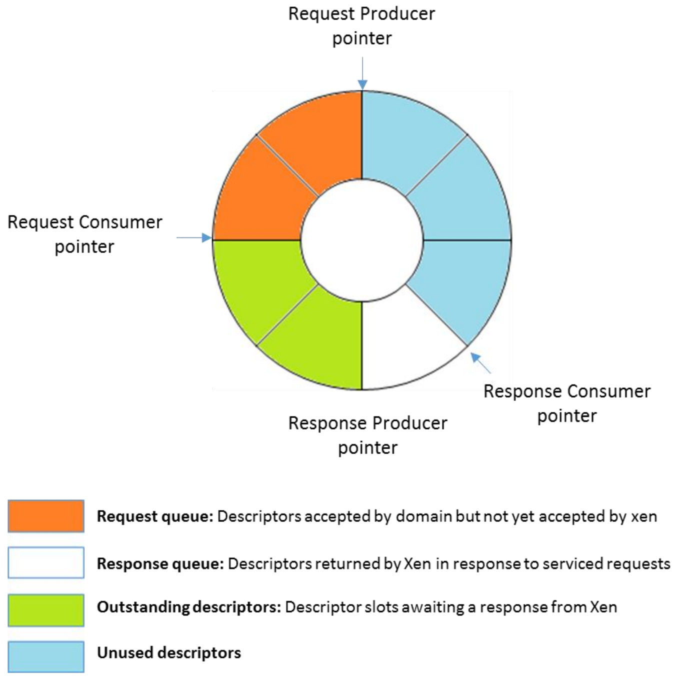

Request Producer pointer
Request Consumer pointer
Response Consumer pointer
Response Producer pointer
Request queue: Descriptors accepted by domain but not yet accepted by xen
Response queue: Descriptors returned by Xen in response to serviced requests
Outstanding descriptors: Descriptor slots awaiting a response from Xen
Unused descriptors

Figure 2.11: Ring I/O buffer

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

2. Request consumer pointer: Xen removes requests which are pointed by request producer pointer by advancing a request consumer pointer.

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

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

The requests are not required to be processed in an 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.

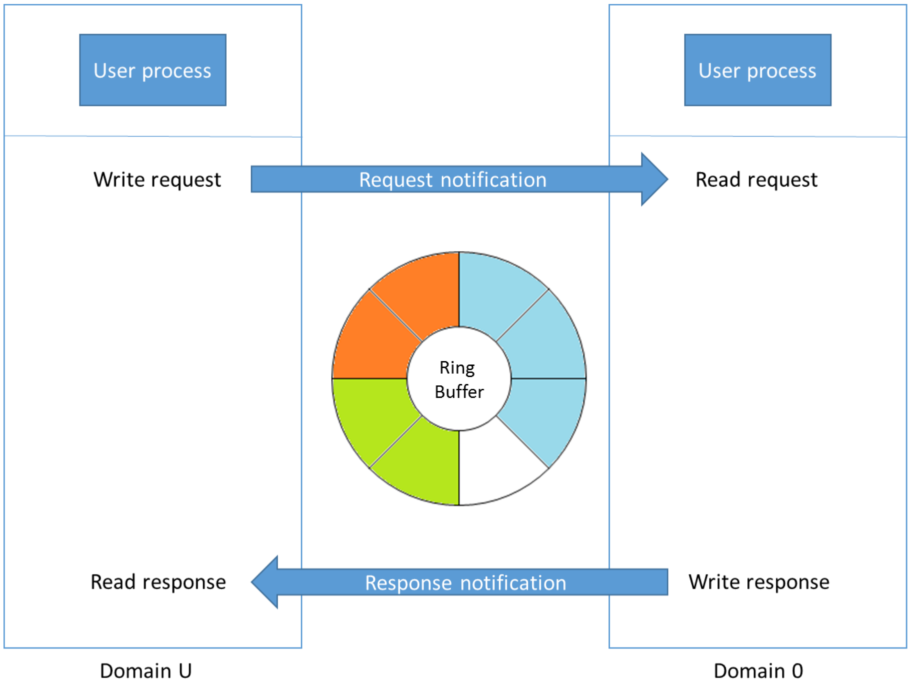


Figure 2.12: Ring I/O buffer

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 can be due to the introduction of hypervisor, data copy overhead or overhead of communication between the domains.

Hypervisor layer: The IDDR system uses Xen hypervisor as a virtualization platform. Xen hypervisor is a Type 1 hypervisor. Type 1 hypervisor runs directly on the host's hardware. It runs between the operating system and the hardware. In Xen, the hypervisor runs in ring 0, while guest OS runs in ring 1 and applications run in ring 3. For hardware access, the guest OS has to go through the hypervisor, which degrades the performance of the system

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 domain executing user applications 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

The 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 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 structure usually results in a large number of broken applications. For 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 [26], which broke the large number of applications. In order to provide compatibility with applications, many microkernel architectures provided an emulation layer [26] for the OSes. The IDDR system maintains compatibility between existing device drivers and applications.

3.3 System overview

Figure 3.1 shows the architectural overview of the modern operating system with a monolithic kernel and the architectural overview of the IDDR system is presented in Figure 3.2.

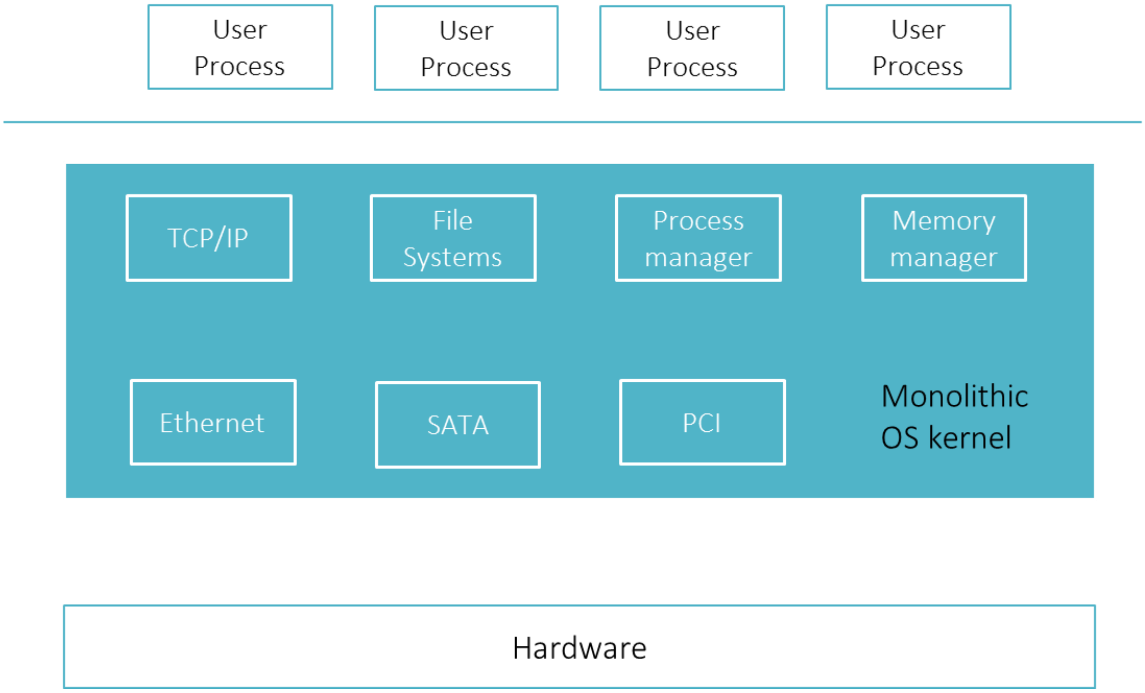


Figure 3.1: Architectural overview of modern OS

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

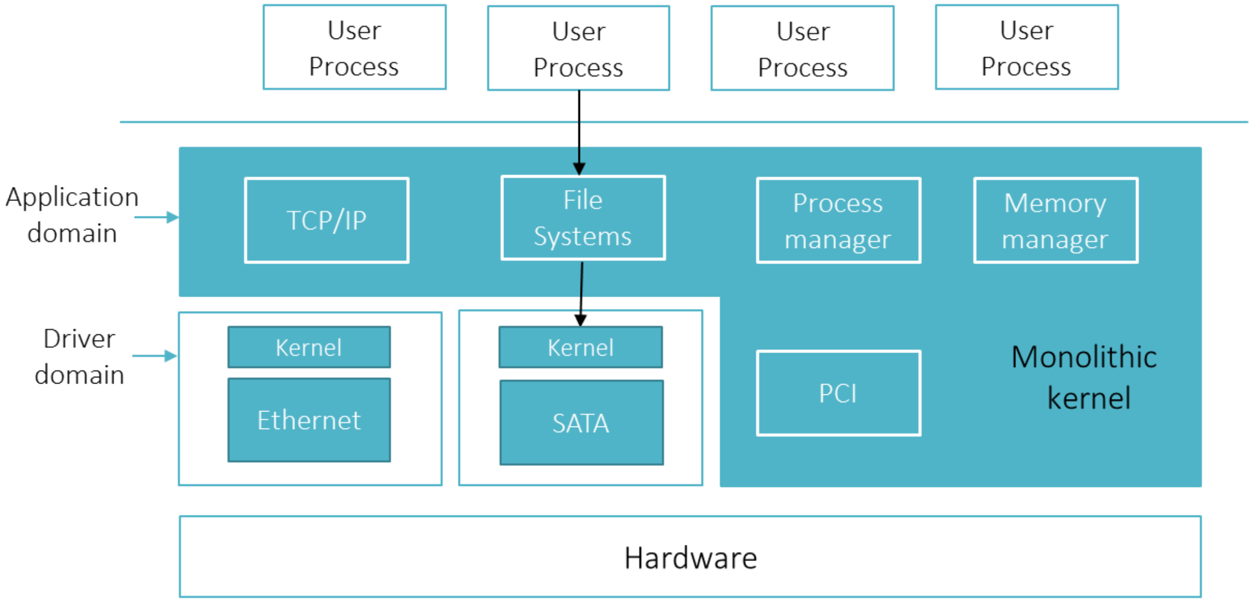


Figure 3.2: Architectural overview of the base IDDR system

same hardware with the help of a VMM. User applications or kernel components access the hardware through the device driver domain.

As the section 5.1 describe 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 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 in 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.

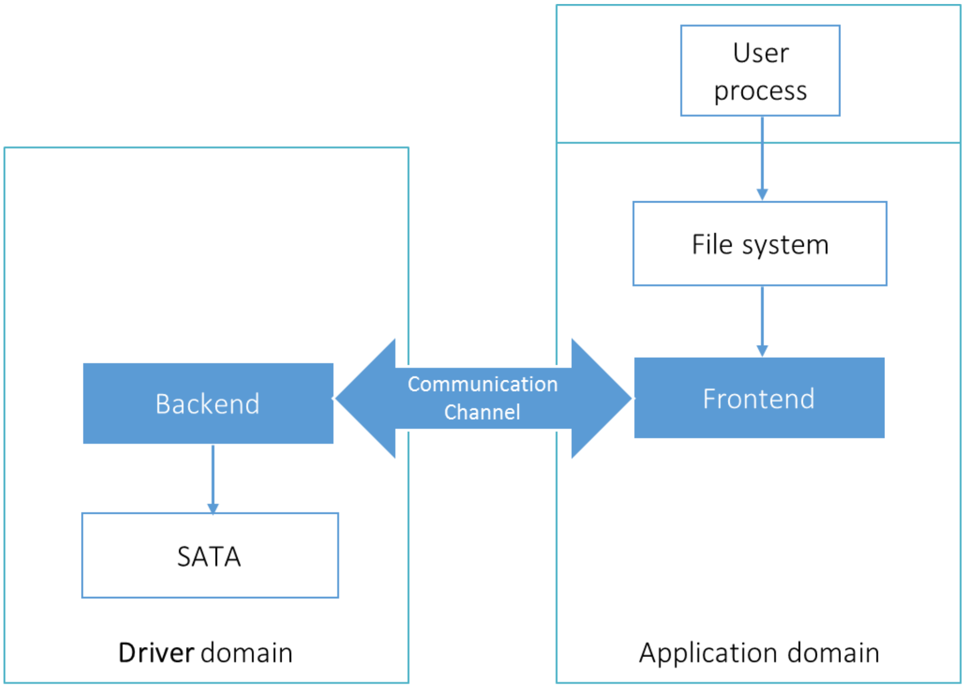


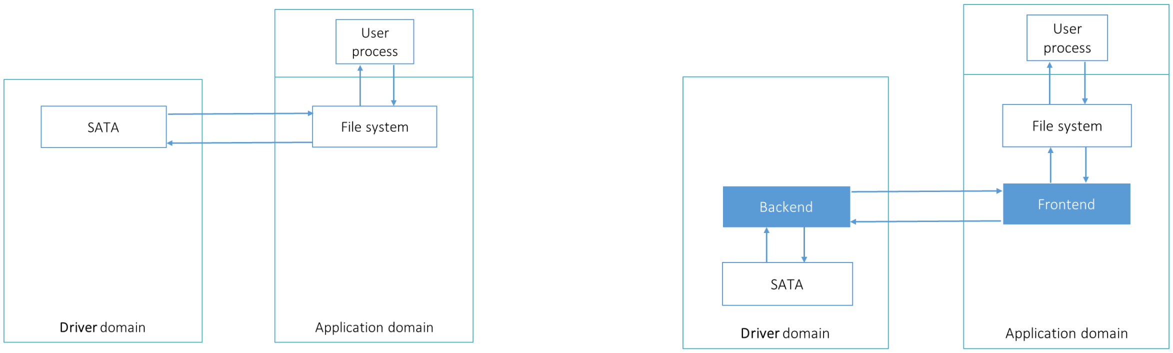
Figure 3.3: System Components

3.4.1 Front end driver

As mentioned earlier in the section 3.2, transparency and compatibility are the properties of the IDDR system, which requires 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. As user applications do not know about the isolated device driver, it is not possible for applications to send requests to the driver in the driver domain, without making any changes to the kernel or the application. The IDDR system runs a piece of a code called *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 driver domain and ends corresponding requests.

3.4.2 Back end 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 canot 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 frontend and 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. The responsibility of the first part is to share the requests and responses between the driver domain and the application domain. The responsibility of the second part is to share the data of read/write requests/responses. 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.

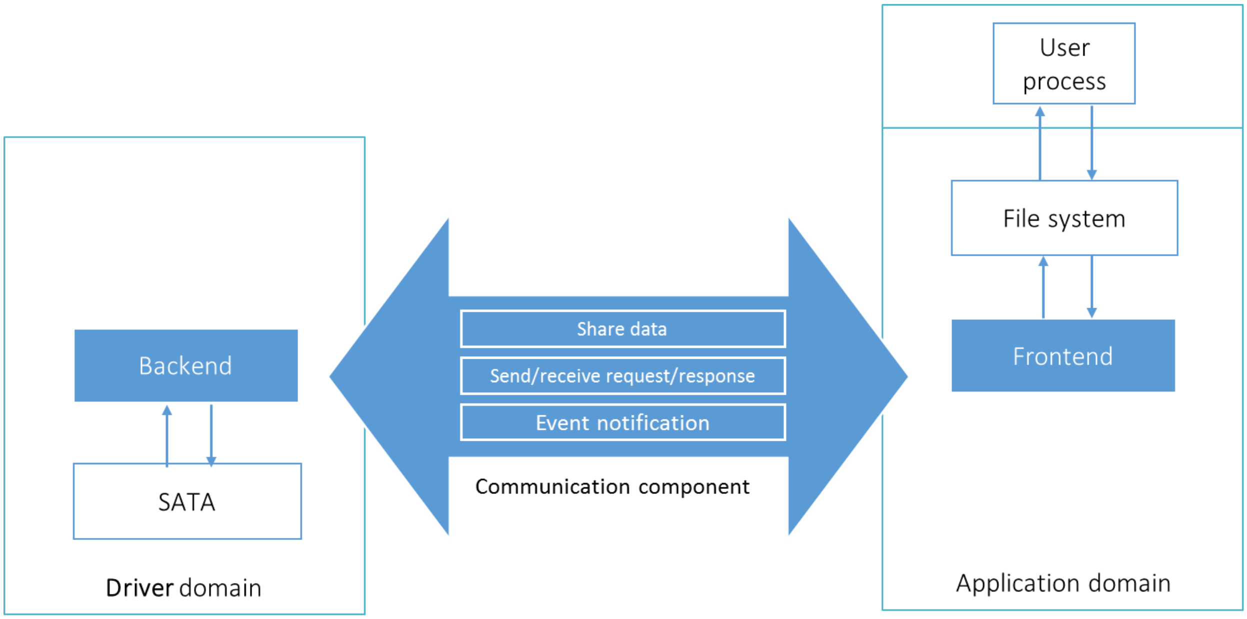


Figure 3.5: Communication module

3.5 System design

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

3.5.1 Communication module

Base IDDR system design In the base IDDR system, the frontend driver submits requests to the communication channel. The communication channel 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.

Spinlock based IDDR system As the Section 3.5.1 describes, in the base IDDR system, the communication module notifies the backend about the avaiability of the 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. These both threads spin for the availability of requests and responses in the shared queue. The intermediate threads delegrates 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 the requests from user application on behalf of the device driver. As explained in the Section 2.2.1, each block device driver has a separate request queue to accept the the requests from user applications. Similar to the block device drivers, the frontend driver also creates a per device request queue to accept requests from user applications. If the sufficient requests are received, the frontend driver flushes the requests to the communication channel. The frontend driver receives a software interrupt upon availability of the 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 requests without accessing the shared memory.

Spinlock based IDDR system As explained in the 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 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 frontend driver main thread and the intermediate thread. The main thread is responsible for reading the requests from the frontend driver request queue (per device 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 introduce a spinlock in the frontend driver. In frontend driver, the main thread spins for the responses for a short time. If a response is available then similar to intermediate thread, the main thread reads the response and shared data 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 for the request to the device driver. Upon completion of the requests, the backend driver reads the response and puts it in shared queue. It also copies the read operation data to the shared memory.

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

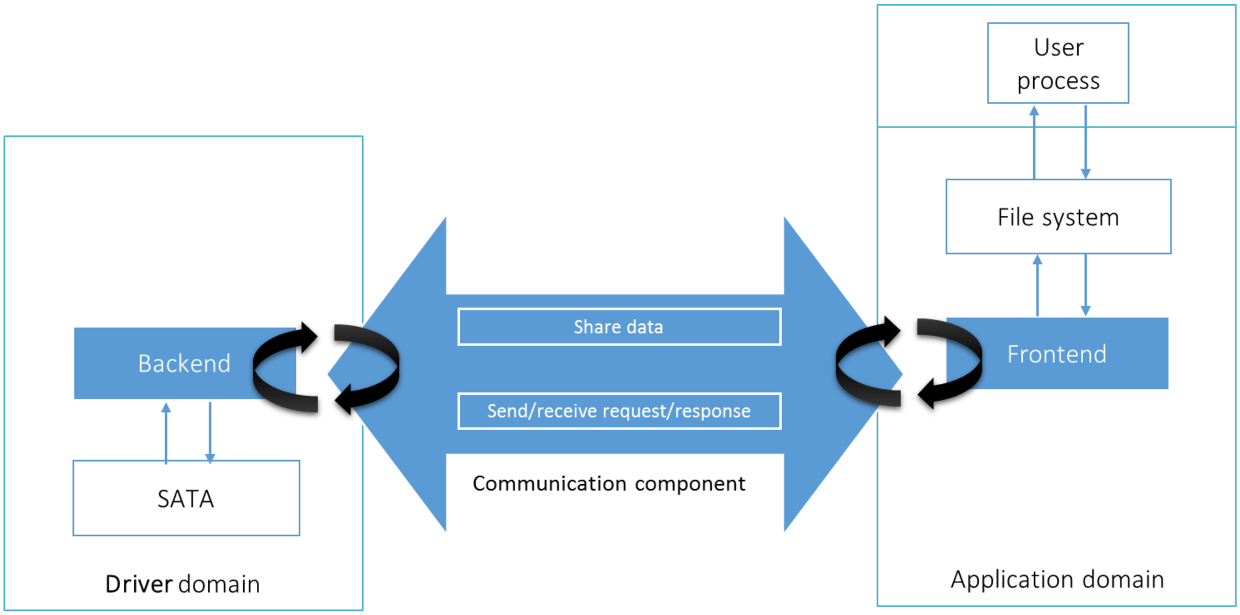


Figure 3.6: Spinlock based new IDDR system

Chapter 4

System Design and Implementation

This chapter describes the specific implementation details of the IDDR and its communication channel.

4.1 Implementation Overview

The IDDR system is implemented with Linux kernel 3.5.0 and Xen hypervisor 4.2.1. For the prototype, we implemented the IDDR system with the isolated block device driver. Both the application domain and the driver domain run the same Linux kernel. The following table summarizes our implementation efforts of the IDDR system.

Component	Number of Lines
Linux Kernel	7
Xen	250
Front-end Driver	647
Back-end Driver	752
Total	1656

As we observe from the table, the block device driver is unchanged, small number of changes were made to the Linux kernel and the Xen hypervisor. However, we maintain the application compatibility.

4.2 Implementation

4.2.1 Communication component

The most important component in the IDDR system implementation is the communication component.

This section will describe the implementation details of the communication channel of the IDDR and the implementation details to improve performance of the communication channel.

1. In the original implementation of the IDDR, communication channel uses event channel for notifying driver domain and application domain when a request or response is available in the shared request and response queue.

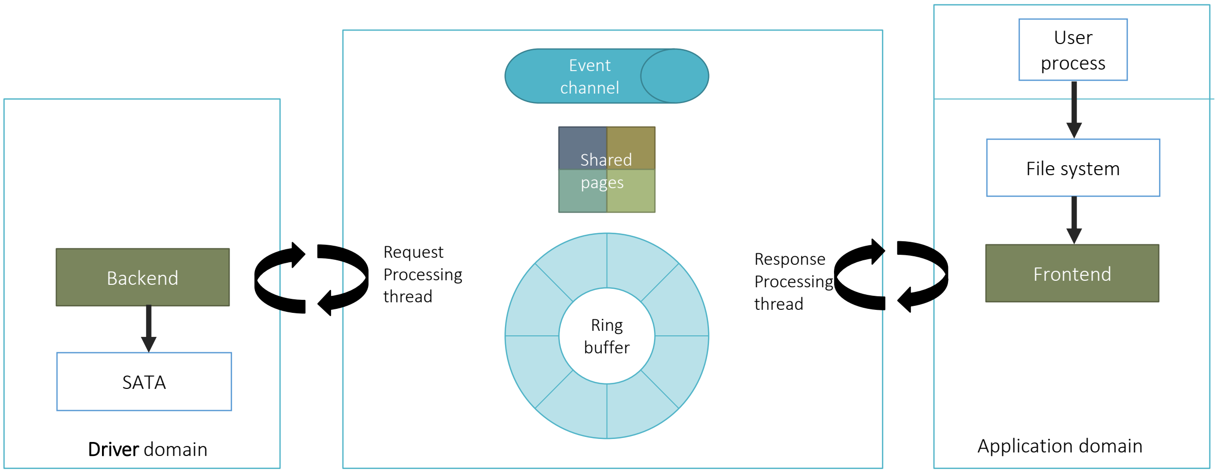


Figure 4.1: Implementation overview

2. In order to improve the performance of the IDDR system, we implement the communication channel in which the front end driver thread spins for the availability of responses, and a dedicated thread in the back end driver spins for requests. In case of unavailability of requests and responses, threads go to sleep. In this implementation event channel is used only to wake these threads up from the remote domain.

The following subsections describe the implementation details of the communication channel with and without performance improvement measures.

Ring buffer: I/O rings

In the implementation of communication channel, for both approaches, ring buffer is used as a request and response queue. Ring buffer is a shared I/O ring explained in section 2.4.2. A ring buffer is divided into front ring and back ring. The front ring is used as a request queue and the back ring is used as a response queue. The front end driver recieves the request from an application, and converts the request into a format which can be understood by the back end driver. The front end device driver then checks for a free space in the request queue for a new request, and allocates the space for the new request using the function `RING_GET_REQUEST`.

After batching sufficient requests together, the front end device driver pushes the requests to the front ring using the function `RING_PUSH_REQUESTS_AND_CHECK_NOTIFY`

Shared pages

Since the ring buffer is not large enough to hold the read and write data of responses and requests, we use it only for sharing the requests and responses. In order to share an actual data we use 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 [4].

Grant References

Grant references are the entries in the grant table. A grant reference entry has every detail about the shared page, which removes the dependency on the real machine address of the shared page. Since there exists a fully virtualized memory, the biggest difficulty in sharing the memory correctly between domains is knowing its correct machine address. Removing the dependency with the real machine address makes it possible to share the memory between domains.[14, 7, 4]

We use grant table so that the application domain grants the driver domain access to the the shared page, while retaining the ownership. The front end driver grants memory access to the back end driver, so that back end driver may read or write data into the shared memory as requested.

The steps we implement are:

1. The application domain creates a grant access reference, and shares the reference id (ref) to the block device driver domain by enqueueing a request in the request queue (front I/O ring) .

2. The block device driver domain reads the request and the reference id, and uses the reference to map the foreign access granted frame.

3. The block device driver domain performs the memory access.

4. The block device driver domain unmaps the granted frame.

5. The application domain removes its grant.

Implementation details of the above steps is explained below:

1. To grant a foreign domain access, front end domain first claims a grant reference entry in the grant table using function `gnttab_claim_grant_reference`, and after that grants the access to foreign domain using function `gnttab_grant_foreign_access_ref`.

2. Back end driver maps foreign frames using `gnttab_set_map_op` and flags `GNTMAP_host_map`.

3. Back end driver unmaps the foreign frames using function `gnttab_set_unmap_op` and `gnttab_unmap_refs`

4. Front end driver removes foreign access to the frames using function `gnttab_end_foreign_access`. `gnttab_end_foreign_access` does not revoke access; it only prevents further mappings. Since `gnttab_end_foreign_access` does not revoke access it is used after the block device driver has unmapped the frame[14, 7].

Event Channel

Event channel is a mechanism provided by xen hypervisor for event notification. Basic implementation of the IDDR communication channel uses event channel to send notifications between domains that request and response is available in the request and response queue.

Event channel : Hypercall interface

`EVTCHNOP_alloc_unbound` is a hypercall to allocate a new event channel port. Allocated event channel port can be connected by remote domain if,

1. Specied domain exist
2. A free port exist in the specified domain.

Less privileged domains can allocate only their own ports, privileged domains can also allocate ports in other domains[14, 7]. `bind_evtchn_to_irqhandler` is used for assigning an interrupt handler for a notification. In driver domain implementation, back end driver allocates an event channel while initializing, and assigns an interrupt handler.

Event channel: Other interfaces

`bind_interdomain_evtchn_to_irqhandler` is used for connecting to existing event channel as well as assigning an interrupt handler for handling a notification. In driver domain implementation back end driver allocates the event channel, and binds the interrupt handler to allocated event channel using `bind_interdomain_evtchn_to_irqhandler`.

In driver domain implementation, front end driver connects to the event channel allocated by backend, using interface `bind_interdomain_evtchn_to_irqhandler`.

4.2.2 Application domain

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

In Xen driver domain implementation block device runs separately in a driver domain. When user process sends a request to the file system, the file system needs to forward the request to storage domain. Like explained in the section 3.4.1, in the implementation of xen driver domain, a piece of code is introduced which forwards the request to the domU running device drivers.

Front end driver

The piece of code, which forwards the request to the domU running device driver is called as a Front end driver. The core responsibility of the front end driver is:

1. To provide an interface which appears as a block device to upper layer in the stack.

2. Accept a request from the upper layer.

3. Create a new request which can be understood by storage domain.
4. Enqueue new request into request queue.

Implementation details of front end driver is split into 4 stages.

1. Initialization
2. Create request
3. Enqueue request
4. Dequeue response

Initialization

During the initialization process the front end driver creates an interface for all the block devices. The interface for each block driver is associated with a queue. Read and write requests issued on the interface gets enqued in the queue. After creating the interface for all block devices, front end driver creates a kernel thread `read_response_thread`

The core functionality of the `read_response_thread` is to dequeue the responses available in the ring buffer. However, there might be a case when no request is available in the ring buffer, but the request is expected to be present in the future. In such cases, `read_response_thread` thread spins for the responses. The `read_response_thread` thread goes into a sleep state after spinning for some time-threshold.

Obviously, a thread shouldn't sleep unless it is assured that somebody else, somewhere, will wake it up. The code doing the waking up job must also be able to identify the thread to be able to do its job. We use a data structure called a wait queue to find the sleeping thread. A wait queue is a list of threads, all waiting for a specific event[?, 10].

In Linux kernel like all other lists, a wait queue is managed by a wait queue head, of a data type `wait_queue_head_t`, and is defined in < *Linux/wait.h* >. A wait queue head is defined and initialized statically as follows: `DECLARE_WAIT_QUEUE_HEAD(name);` and dynamicly as follows: `wait_queue_head_t my_queue; init_waitqueue_head(&my_queue);`

Create request

Front end driver dequeues the request submitted to the driver interface by a user process or the file system, and then converts the request into a request of structure type `idd_request_t`. The structure is as below:

```
struct idd_request {
  int data_direction;
  uint8_t nr_segments;
  uint64_t sector_number;
  struct idd_request_segment {
      grant_ref_t gref;
      uint8_t first_sect, last_sect;
  }seg[IDD_MAX_SEGMENTS_PER_REQUEST];
  uint64_t seq_no;
}__attribute__((__packed__));
```

Member of the structure are explained below. data_direction : Flag to tag if request is read or write. nr_segments : Number of idd_request_segments. gref : Grant reference grant table entry. first_sect and last_sect : first and last sector in frame to transfer. seq_no : To track if any request and response is lost.

Enqueue request

Like explained in section 4.2.1, RING_PUSH_REQUESTS_AND_CHECK_NOTIFY is used for flushing request to the ring buffer, we use the same API to enqueue the request to the request queue. However, if the read_request_thread running in back end driver which accepts the requests is sleeping, then waking up the read_request_thread is more important task of front end driver. Wake up signal is sent to back end driver using an event channel. Before sending the notification, we check the state of the thread. Status of the thread is saved in the shared memory. We use the atomic variables to save the state of threads to avoid race conditions.

Dequeue response

Response is dequeued from the ring buffer by the read_response_thread. To dequeue the response from response queue, we use the ring buffer API RING_GET_RESPONSE. However, the important part in this stage it managing the read_response_thread thread.

When the response is not available read_response_thread spins for some time, and then goes to sleep. When the response is made available by the backend then depending upon the state of the thread, event channel interrupt is sent by the backend. If `read_response_thread` is in sleeping state then the backend driver will send an interrupt and the front end driver will wake up the thread, otherwise, no action is taken as thread is already spinning for the response.

In interrupt handler shared atomic variable status is read by the front end driver and depending upon state action is taken. `read_response_thread` sleeps on the wait queue, waiting for a flag denoting availability of the response to be set. Once the response is available in the response queue, it is read using ring buffer API `RING_GET_RESPONSE`.

We also maintain an shadow table of all requests submitted to the back end driver. The requests are read from the shadow table and are ended upon successfully reading the respective response. The request is ended by using a function `__blk_end_request_all`

Upon reading the responses, thread spins again for more responses and after reaching the threshold goes to sleep. We mark the state of the thread as `SLEEPING` and then check for the request queue to avoid race condition. If a request is present in the request queue then the request is served while state of the thread is still `SLEEPING`.

4.2.3 Driver domain

Driver domain is the domU running a device driver. In our implementation, driver domain runs block device driver. Usually in monolithic Linux kernel an user process sends the read write request to a file system, which sends the read and write request to block device driver.

Block device driver serves the request and responses back to file system, which further sends response to user process.

However, in driver domain implementation, block device runs separately in a driver domain. Like explained in a section 3.4.2, a piece of code called as a back end driver runs in a driver domain which accepts a request from application domain and forward the request to the device driver. Upon receiving the response from the device driver, back end driver sends back the response, and notifies the application domain.

Back end driver

Back end driver is a kernel module and component of IDDR which runs in the driver domain. The core responsibility of back end end driver is :

1. Dequeue request from request queue.
2. Convert to BIO request which can be understood by block device driver.
3. Accept response from block device driver.
4. Enqueue response into response queue.

Implementation details of back end driver can be split into 5 stages.

1. Initialization
2. Dequeue request

3. Create BIO.
4. Make response.
5. Enqueue response

Initialization

During initialization process Back end driver creates a kernel thread `read_request_thread`. The core functionality of the `read_request_thread` is to dequeue the requests available in the request queue. If request is not available in the request queue then thread waits on a wait queue. Similar to front end, back end driver initializes wait queue in initiazation process.

Dequeue request

Request is dequeued from the request queue by the `read_request_thread`. To dequeue a request, we use the ring buffer API `RING_GET_RESPONSE`. When the request queue is empty, `read_request_thread` spins for some time to checks if new requests are queued, after reaching threshold it goes to sleep. When the request is enqueued by front end driver then depending upon the state of the `read_request_thread`, an event channel interrupt is sent by the front end driver. If `read_request_thread` is in a sleeping state then front end driver will send an interrupt. In intrupt handler, back end driver will wake up the `read_request_thread`. If `read_request_thread` is already running then no action is taken. In interrupt handler, status is read from shared atomic variable by the back end driver.

read_request_thread sleeps on a wait queue waiting for a flag to be set. The flag denotes the avilability of the request in the request queue. The request is read from the request queue using ring buffer API RING_GET_REQUESTS.

After reading the requests, read_request_thread thread spins again for more requests. If a threshold limit is reached then the read_request_thread thread goes to sleep. We mark the state of the thread as SLEEPING and then check for the request queue one more time to avoid a race condition. If request is present in request queue then that request is served while state of the thread is still SLEEPING.

Create BIO

Whenever the request thread receives a request to serve, the request thread creates the bio request for the corresponding request. bio structure is a basic container for block I/O within a kernel. bio structure is defined in $< Linux/blk_types.h >$. bio structure represents active block I/O operations as a list of segments, and a segment is a chunk of buffers. The bio structure provides the capability for the kernel to perform block I/O operations of even a single buffer from multiple locations in memory. Vector I/O such as this is called scatter-gather I/O.

A request queued into the request queue by the front end driver is in format which is understood by back end driver, but we cannot forward the same request to block device. We convert the dequeued request into bio request, so that the block device understands the request. In order to make the bio request, we need the associated block device structure struct block_device. We get associated block device structure using function blkdev_get_by_path. Pages from shared memory are mapped and inserted into the bio structure using function bio_add_page. And other variables are copied into the bio structure from the dequeued request. At the end, the newly created bio request is sent to the lower layer for execution with submit_bio. Once bio request is completed, function pointed by a function pointer bi_end_io gets called.

Make response and Enqueue

Irrespective of the success or failure of the execution of bio request, the back end driver makes a response, which could be understood by the frontend. Like explained in subsection 4.2.3, bi_end_io function pointer is a pointer to a callback function. Once bio request is completed, function pointed by bi_end_io gets called. We create a new response in this callback function.

In this callback function we complete the bio request with function bio_put. After that the result gets copied into a newly allocated response structure. The response is enqueued to response queue using RING_GET_RESPONSE and RING_PUSH_RESPONSES_AND_CHECK_NOTIFY. Once the response is enqueued, depending upon the status of the remote thread-read_response_thread, a interrupt signal is sent to the application domain.

Chapter 5

Evaluation

The IDDR implementation uses the `Linux kernel 3.5.0` for both application domain and driver domain. We tested it with Arch Linux on `x86_64` platform. The specification of the system used for evaluation is presented in the table 5.1.

5.1 Goals

Our evaluation contains

1. comparison of the Xen's isolated driver domain with baseline IDDR system,
2. an evaluation of IDDR performance improvement over base IDDR system,

The first goal of the evaluation is to verify and compare the performance of the baseline IDDR system with that of the Xen's isolated driver domain. The comparison shows that

Table 5.1: Specifications of the system

System Parameter	Configuration
Processor	1 X Quad-code 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 ??RPM

the performance of the IDDR system matches the performance of the Xen's isolated driver domain.

The second goal is to show that the performance of the IDDR system improves if a frontend and backend driver spins over a spinlock to check the availability of requests and responses, instead of sending event channel interrupts to notify the availability of requests and responses.

5.2 Methodology

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. We use block devices such as SATA disk, ramdisk and loop device for the performance testing. These devices cover the variety of block devices we would want to test our system against.

In order to measure performance, we format the block device with the ext2 file system, and run the fileIO SysBench benchmark [5] 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. SysBench can run a specified number of threads by executing all requests in parallel. Sysbench benchmark in fileIO test mode generates 128 files with `1Gb` of total data and performs random reads, random writes and mix of random read-writes with a block size of `16Kb`.

5.3 Xen split driver vs IDDR

The IDDR system is a re-implemention of the Xen's isolated driver domain. We improve the performance of the base IDDR system by introducing spinlocks instead of the event channel interrupts in the communication channel. As per our first goal of evaluation mentioned in Section 5.1, we compare the baseline IDDR with the Xen's isolated driver domain. In this comparision, we show that the performance of the base IDDR system matches the Xen's isolated driver domain. This verifies the baseline performance of the IDDR.

As explained in chapter 2, Xen's isolated driver domain follows the architecture of the Xen split driver. In order to measure the performance of the Xen's isolated driver domain, we run performance benchmarks on a block device which uses the Xen split driver.

5.3.1 Experimental setup

Xen split driver

We create a ramdisk in domain 0. The guest domain domU is configured such that the ramdisk uses a split device driver and the ramdisk is available in the guest domain. We do a similar setup for the loop device and the SATA disk. For example, in case of loop device, we create a loop device in 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 5.1.

IDDR

In the Xen split device driver setup, the backend device driver runs in a domain 0 and the frontend device driver runs in the domain U.

Xen paravirtualized guests are aware of the VMM and require special ported kernel to run on Xen VMM, so the guests can run efficiently without emulation or virtual emulated hardware. Paravirtualization does not require virtualization extensions from the host CPU.

Fully virtualized or Hardware Virtual Machine (HVM) guests 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 are used to boost performance of the emulation. Fully virtualized guests do not require special kernel. In order to boost performance fully virtualized HVM guests use special paravirtual device drivers to bypass the emulation for disk and network IO.

Our setup is configured such that the domain U is a HVM guest and the domain 0 is a PV guest. HVM guests are expected to have less syscall overhead and faster memory bandwith than PV guests. In order to have a fair comparision, it is necessary to run the backend driver of the IDDR system in the domain 0 and the frontend driver of the IDDR system in the domain U.

We insert a ramdisk and the IDDR frontend module in the domain 0. The IDDR backend module is inserted in the guest domain domU. We format and mount the ramdisk with ext2 file system and sysbench benchmark is run on it.

Similar setup is used for loop device and SATA disk.

Comparision

Figure 5.1 shows the performance of random reads-writes on a ramdisk, and Figure 5.2 shows the performance of random reads on a loop device. Both systems provide roughly similar performance. The figures show that the performance of the IDDR matches the performance of the Xen's isolated driver domain.

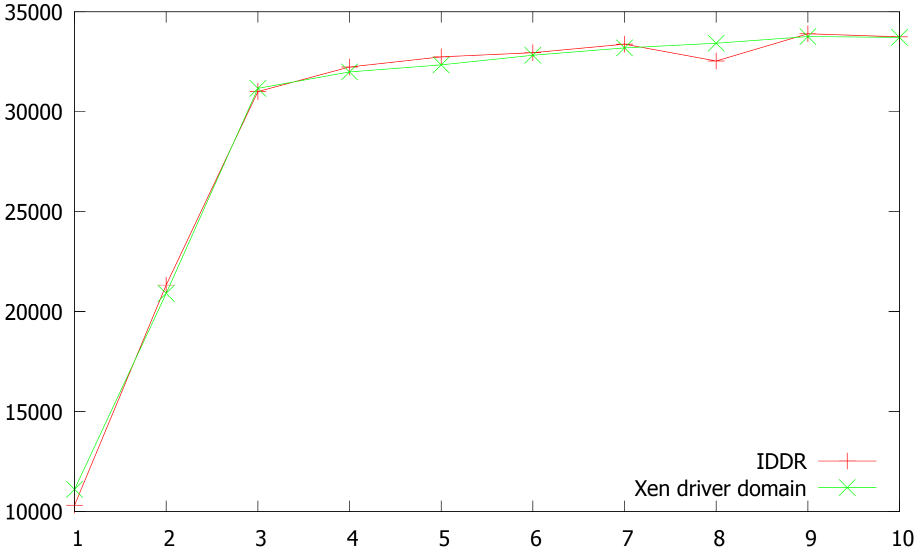


Figure 5.1: IDDR vs Xen split driver

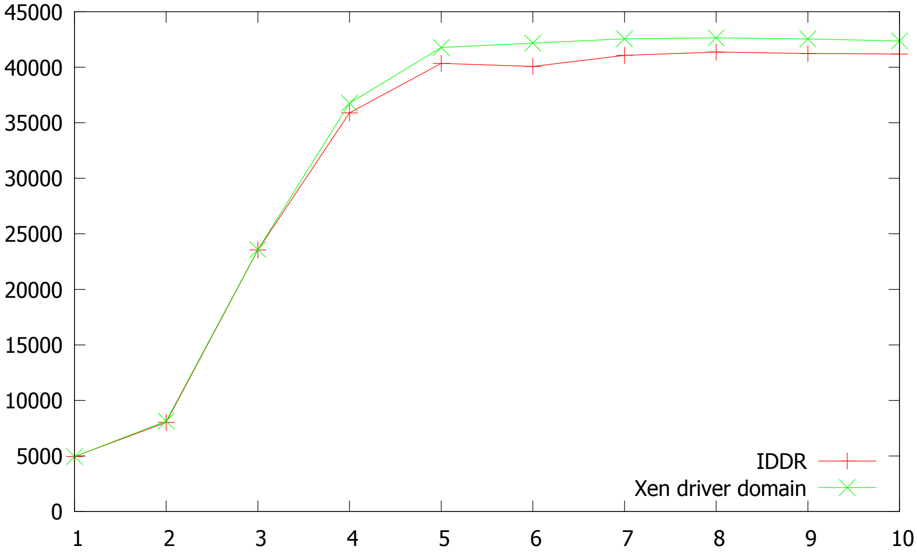


Figure 5.2: IDDR vs Xen split driver

5.4 IDDR performance improvement

We measure and compare the performance of the base IDDR system (event channel interrupt) with the new IDDR system with spinlock. We run the fileIO sysbench benchmark with random reads, random writes and mixed random reads-writes.

5.4.1 Experimental setup

In the setup, domain 0 is an application domain, and domain U is a driver domain. A ramdisk is created in the domain U (driver domain). The backend driver is inserted in the domain U (driver domain) and the frontend driver is inserted in the domain 0 (application domain). The disk is formatted and mounted with ext2 file system in the application domain. Sysbench benchmark is run on the mounted partition.

For loop device a similar setup is used where we create a loop device in a driver domain, and insert the backend driver in the driver domain. The front end driver is inserted in domain 0. For SATA disk, we passthrough SATA disk to driver domain, so that the driver domain can directly access the SATA disk.

Comparision

Figure 5.3 compares the performance of the base IDDR and the new IDDR with random reads-writes on a ramdisk. The figure shows that the new IDDR system implementation with spinlock performs better than the base IDDR implementation.

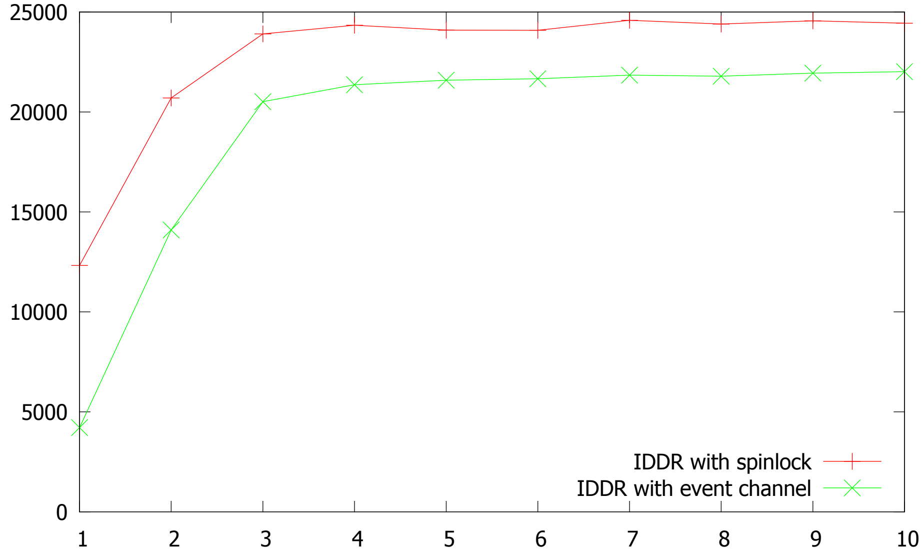


Figure 5.3: IDDR with Spinlock vs IDDR with event channel

Chapter 6

Related Work

This chapter first presents existing inter domain communication approaches. Subsequently, the chapter presents work which improves the performance of inter domain communication.

In the past numerous work on inter domain communication mechanisms was presented. Xen split drivers is one of the inter domain communication approach of Xen hypervisor [21]. The xen split drivers has overhead becuase of numerous context switches in form of event channel interrupts. It also incurs an overhead due to data copy, page flipping [44]. Xen hypervisor also provides a UNIX domain socket like interface for high throughput interdomain communication on the same system called XenSocket [44]. XenSocket replaces the page flipping design of the split driver. However, XenSocket needs an existing socket interface APIs to be changed.

Fido [12] is a shared memory based inter domain communication mechanism. Fido implements the fast interdomain communication mechanism by reducing data copies in Xen hypervisor. In contrast our system improves the inter domain communication mechanism of Split device drivers by avoiding the context switches.

VirtuOS [34] is a library level solution that allows processes to directly communicate with domain.

Chapter 7

Conclusion and Future Work

7.1 Contributions

The idea is make the system general enough to support multiple disaster relief studies.

Bibliography

[1]

[2] Coverity - Linux kernel report. http://www.coverity.com/library/pdf/coverity_linuxsecurity.pdf.

[3] hypercall. http://wiki.xen.org/wiki/Hypercall.

[4] hypercall. http://xenbits.xen.org/docs/4.2-testing/misc/grant-tables.txt.

[5] Sysbench performance measurement benchmark. http://sysbench.sourceforge.net/.

[6] 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.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In

Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[9] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Commun. ACM*, 27(1):42–52, January 1984.

[10] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[11] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[12] 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 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 25–25, Berkeley, CA, USA, 2009. USENIX Association.

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

[14] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[15] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

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

[17] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.

[18] Simon Crosby and David Brown. The virtualization reality. *Queue*, 4(10):34–41, December 2006.

[19] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

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

[21] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[22] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. In *In 9th SIGOPS European Workshop*, pages 109–114, 2000.

[23] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, New York, NY, USA, 1973. ACM.

[24] G. Scott Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[25] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166), February 2008.

[26] Gernot Heiser and Volkmar Uhlig. Are virtualmachine monitors microkernels done right. *Operat. Syst. Rev*, 40:2006, 2006.

[27] Samuel T. King and et al. Operating system support for virtual machines.

[28] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[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 Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 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*, ExpCS '07, New York, NY, USA, 2007. ACM.

[31] Daniel A. Menasc. Virtualization: Concepts, applications, and performance modeling, 2005.

[32] 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*, ATEC '06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

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

[34] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132, New York, NY, USA, 2013. ACM.

[35] year = 2013 Nikolaev, Ruslan and Back, Godmar, title = Design and Implementation of the VirtuOS Operating System.

[36] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM.

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

[38] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, July 2004.

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

[40] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[41] 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*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[42] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

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

[44] 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*, Middleware '07, pages 184–203, New York, NY, USA, 2007. Springer-Verlag New York, Inc.