Technische Universitt Berlin

Fakultt Elektrotechnik und Informatik
Institut fr Softwaretechnik und Theoretische Informatik

Security in Telecommunications

Fakultt IV
Sekr. TEL17
Ernst-Reuter-Platz 7
10587 Berlin
http://www.eecs.tu-berlin.de

Master Thesis

Porting the Xen Emulation Layer to static hypervisor

Amna Waseem

Matriculation Number: 387424
01.11.2017
Supervised by
Prof. Dr. Jean-Pierre Seifert
Assistant Supervisor
Dr.-Ing. Jan Nordholz and Robert Buhren

Abstract

Over the past decade, virtualization technologies have gone from server consolidation in corporate data centers to small forms of embedded platforms to provide system security, hardware isolation, resource management and I/O device emulation. With the modern computers providing different forms of I/O devices, there has been huge development in hypervisors' technology to provide efficiency in the use of physical resources. Major hypervisors in virtualization market have been continuously deploying dfiferent memory techniques to dynamically allocate memory and use resources more efficiently. However, this dynamicity in memory allocation introduces difficulty in provabilty and verification of configured hypervisor used in safety critical applications. PHIDIAS, a static hypervisor developed by Security in Telecommunications department of Elektrotechnik und Informatik faculty at Technischen Universitt Berlin, is built around the concept of Principle of Staticity to ease provability and reduce runtime complexity along with memory footprint by eliminating dynamic elements from the system. However, it does not have support of I/O device emulation. In this thesis, Xen Hypervisor's I/O device framework has been ported to PHIDIAS using existing mechanisms of static memory sharing and inter-guest communications.

Contents

List of Figures

List of Tables

1 Introduction

In this section, a brief introduction of virtualization and hypervisor technologies will be given. A little comparison of Xen, KVM and Phidias will be provided as well.

1.1 Motivation

In this section, motivation behind adding device I/O emulation support in Phidias will be discussed e.g. to prove flexibility of a static hypervisor and to ease provability. This section will also highlights the reasons for choosing Xen as a reference.

1.2 Objective

This section will describe the problems and issues which will be solved in this thesis. The main aim of this thesis will be given in this section e.g using existing Phidias memory sharing and IPI mechanism to port Xen Split Driver model for I/O virtualization and keep changes as minimal as possible.

1.3 Scope

In this section, method for porting device I/O emulation of Xen to Phidias will be discussed i.e. main approach will be discussed to port split driver model to Phidias.

1.4 Outline

This section gives a brief introduction into the main chapters.

This example thesis is separated into 7 chapters.

Chapter 2 is called 'Background'. Here I will give background information to lay foundation of thesis topic.
Chapter 3 provides the requirements and assumptions of thesis

Chapter 4 discussed 'Design' and 'Model'. Here I will describe my approach, give a high-level description to the architectural structure and to the basic components that my design consists of.

Chapter 5 describes the implementation part of my work.

Chapter 6 is 'Evaluation'. How the testing is performed for thesis will be discussed. Measurements, tests, screenshots will be included.

Chapter 7 summarizes the thesis, describes the problems that occurred and gives an outlook about future work.

2 Background

In order to set context of underlying work of thesis, this section provides background information on virtualiaztion technologies and embedded systems with special emphasis on the ARM platforms. A brief overview of Phidias, a static hypervisor in question is also given in this section.

2.1 Introduction to Virtualization

Virtualization is a mechanism of providing abstraction between computer hardware systems and softwares running on them, allowing us to create multiple computing environments exploiting the resource isolation on a single physical platform. It basically gives a logical view of multiple operating environments running on a single hardware. With the recent developments in virtualization, there has been huge investments in organizations over this technology to improve the efficiency and availability of resources and applications. Enterprises are giving up the old **one server, one application** model and gaining benefits from server consolidation provided by virtualization. Virtualization has dramatically changed the IT landscape by reducing its expenses and providing economies of scale and greater efficiency.

2.1.1 History of Virtualization

History of virtualization dates back to 1950's when the Compatible Time Sharing System (CTSS) was developed at MIT on IBM 704 series computer. The supervisor program of CTSS handled console I/O, scheduling of foreground and background (offline-initiated) jobs, temporary storage and recovery of programs during scheduled swapping, monitor of disk I/O, etc. The supervisor had direct control of all trap interrupts [11].

In the fall of 1963, MIT's Project MAC was founded with the main purpose of designing and implementation of a better time sharing system than CTSS. After IBM lost the bid to General Electric's GE 645, it created a number of virtual machine systems e.g, the CP-40 (developed for a modified version of IBM 360/40), the CP-67 (developed for the IBM 360/67), VM/370, and many more. We can roughly say that Virtual Machine technology was brought to users with the introduction of the CP-67 hypervisor on the S/360 Model 67 processor. In 1999, VMware introduced virtualization on x86 platforms. Since then, many vendos like Microsoft, Citrix etc had followed VMware and technology has been evolved with the advances in hardware architectures.

2.1.2 Benefits of Virtualization

For many years, server virtualization was considered one of the biggest advantages of using virtualization technology and VMware enjoyed a long run as king of x86 server virtualization. However, many players e.g. Citrix and Microsoft started to gain ground in this field by providing additional middleware and desktop virtualization offerings. Over the past several years, there has been huge deployments in virtualization and vendors are continuously innovating to increase virtualization capabilities of systems and developing management tools.

There are many advantages of using virtualization technology. Following are some of the main benefits due to which virtualization has become a mainstream tool in the computing industry [12]:

Better utilization of resources

With virtual machines, resources of computing platforms can be used in an optimal manner to achieve better performance. Servers used in data centers typically have large resource capabilities. However, they are not fully utilized because of small number of connected users or less demanding applications. Virtualization of hardware allows on-demand resource allocation leading to efficient use of computing power, storage space and network bandwidth. In addition to on-demand usage of resources, virtualization also provides resource isolation between virtual machines. Each virtual machine can run software without affecting others code execution.

Consolidation

For many years, individual servers have been dedicated to run single applications. For less demanding applications, computing capabilities would be wasted. With the advent of virtualization, organizations are now deploying several applications on single servers using only a small amount of processing power. Server consolidation has led to dramatic reduction in need of floor space, HVAC, A/C power, and co-location resources which has caused cost reduction and efficient power consumption.

Security and Isolation

Virtualization allows running multiple virtual machines in an isolated secure environment. All privileged calls made by guests' kernels are analyzed by hypervisor to provide safety against vulnerabilities and attacks. Exceptions and traps of one virtual machine are handled by hypervisor layer isolating other virtual machines from the resulting affects. Virtualization regulates access permissions to programs with reduced privileges from misusing resources.

Migration and Increased Uptime

Migration is the process of moving a running virtual machine from one place to another without affecting overall system. With virtualization, organizations can get better performance and reliable systems. It also increases uptime of servers and applications. Virtual machines can easily be backed up and restored for speedily recovery from computing disasters.

2.1.3 Overview of Hypervisors

A virtualization layer that separates the service request from underlying physical delivery of that request is called virtual machine monitor (VMM) or hypervisor. Hypervisor allows multiple operating systems to run concurrently within virtual machines on a single computer and provides dynamic allocation and sharing of physical resources e.g. CPU, memory, storage and I/O devices [13].Hypervisor enables communication between hardware and a virtual machine so that the virtualization accomplishes with this abstraction layer (hypervisor) [14]. Figure 2.4 shows the virtual machine abstraction architecture.

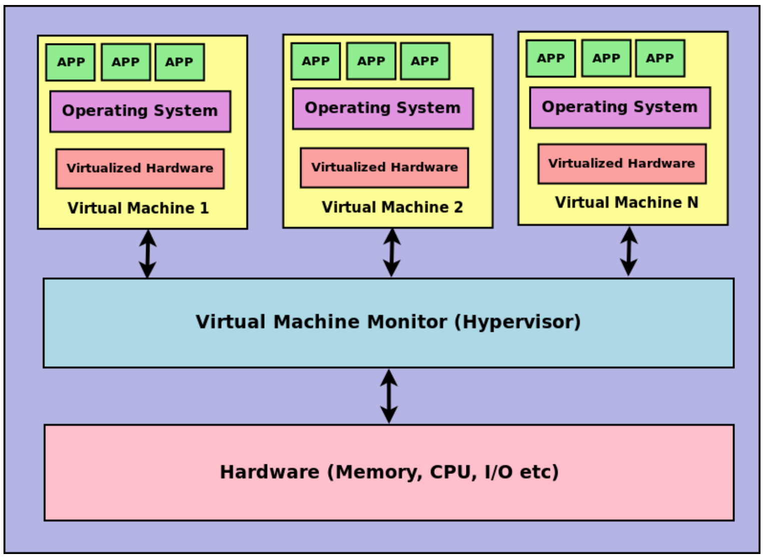


Figure 2.1: Virtualization Framework

In 1974, Popek and Goldberg described the requirements of a hypervisor for efficient virtualization in the article 'Formal Requirements for Virtualizable Third Generation Architectures' [15]. There are three requirements to be fulfilled by hypervisors:

- Virtualization environment provided by hypervisor should be native system so that program behaves in a similar fashion.
- Virtualized resources should be shared with security controls to protect from any threats and performance interference.

- Good support to handle privileged instructions should be provided in order to avoid performace degradation.

Keeping in view the above requirements, different types of hypervisors have been introduced in market with different implementation level of virtualization which will be described in next sections.

2.1.4 Types of Hypervisors

There are two basic types of hypervisors i.e. Type 1 and Type 2 Hypervisors as shown in Figure [**?**].

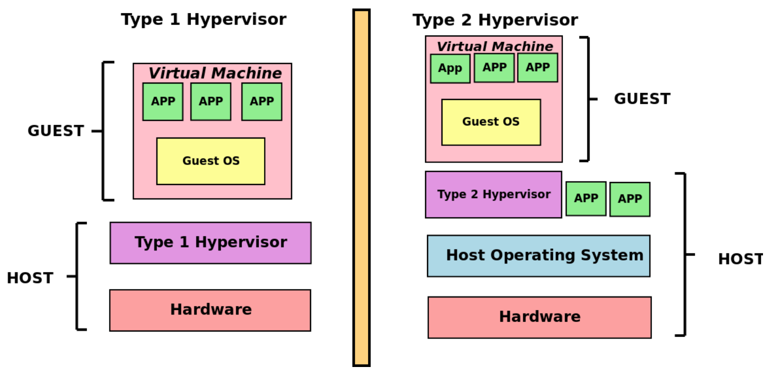


Figure 2.2: Type1 Hypervisor vs Type 2 Hypervisor

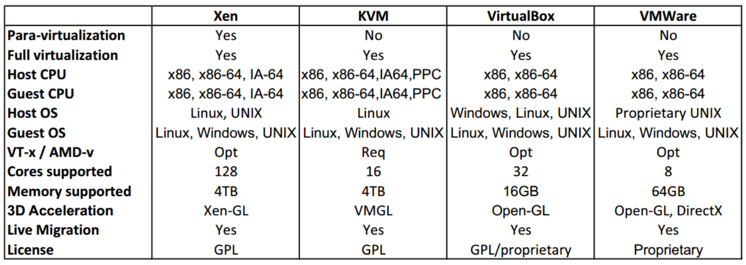
- **Type 1** hypervisor sits directly on hardware and manages virtual machines on top of it. It is also called bare-metal hypervisor. Examples of type 1 hypervisors are VMware vSphere/ESXi, Microsoft Windows Server 2012 Hyper-V, Citrix XenServer, Red Hat Enterprise Virtualization (RHEV) and open-source Kernel-based Virtual Machine (KVM) [14].

- **Type 2** hypervisor runs on host operating system to manage virtual machines here hosted operating system provides hardware configuration. VirtualBox and VMware Workstation are examples of type 2 hypervisors.

According to IBM, Type 1 hypervisors provide higher performance, availability, and security than Type 2 hypervisors.IBM recommends that Type 2 hypervisors be used mainly on client systems where efficiency is less critical or on systems where support for a broad range of I/O devices is important and can be provided by the host operating system [16].
Since bare-metal hypervisor has direct access to the hardware resources rather than going through an operating system, it is considered to be more efficient than a hosted architecture and delivers greater scalability, robustness and performance [13].

2.1.5 State of the Art Hypervisors

Over the past decade, virtualization technology has gone from small deployments to full blown IT infrastructure development. Technology makers have shifted their focus from operating systems with one-to-one relationships with hardware to virtualized approaches to shared resources among multiple operating systems on one hardware. When we talk about virtualization players in market, VMware is the first one which comes to our mind. Besides VMware, now other players like Citrix XenServer, Microsoft Hyper-V,Red Hat Enterprise Virtualization (RHEV), Oracle's Solaris Zones, LDoms and xVM, Amazon's Elastic Compute Cloud (EC2), Google Ganeti cluster virtual server management software tool and Virtual Bridges' VERDE product [17] have entered this new market and are continuously developing innovation virtualization solutions to meet specific requirements of different industries. In 2011, Younge, Andrew J., et al. [18] compared several virtualization technologies which is provided in Figure 2.3.



	Xen	KVM	VirtualBox	VMWare
Para-virtualization	Yes	No	No	No
Full virtualization	Yes	Yes	Yes	Yes
Host CPU	x86, x86-64, IA-64	x86, x86-64,IA64,PPC	x86, x86-64	x86, x86-64
Guest CPU	x86, x86-64, IA-64	x86, x86-64,IA64,PPC	x86, x86-64	x86, x86-64
Host OS	Linux, UNIX	Linux	Windows, Linux, UNIX	Proprietary UNIX
Guest OS	Linux, Windows, UNIX	Linux, Windows, UNIX	Linux, Windows, UNIX	Linux, Windows, UNIX
VT-x / AMD-v	Opt	Req	Opt	Opt
Cores supported	128	16	32	8
Memory supported	4TB	4TB	16GB	64GB
3D Acceleration	Xen-GL	VMGL	Open-GL	Open-GL, DirectX
Live Migration	Yes	Yes	Yes	Yes
License	GPL	GPL	GPL/proprietary	Proprietary

Figure 2.3: A comparison chart between Xen, KVM, VirtualBox, and VMWare ESX. Adapted from from "Analysis of virtualization technologies for high performance computing environments" by Younge, Andrew J., et al, 2011, p11

2.2 Embedded Systems

Embedded systems are composed of simple devices used to perform small and dedicated functions with specific hardware and software constraints like low memory and efficient power usage, more battery life, smaller code footprint, smaller size and weight with reduced cost and reliability [19]. According to Steve Heath [20], an embedded system is a microprocessor-based system that is built to control a function or a range of dedicated functions and is not designed to be programmed by the end user in the same way as general purpose PC is. He had described several features of embedded systems which had led to the wide spread use of microprocessors in industry. Some of main features are as follows:

- **Replacement of discrete logic circuits** to reduce the time and cost of developing new products by changing program code to process data in embedded systems.

- **Upgrading systems** by changing the software while keeping the hardware same thus reducing the cost of production and testing of software.

- **Providing easy maintenance upgrades** for adding new functionalities and resolving bugs by reprogramming the software without modifying the hardware.

- **Protecting intellectual property** by encapsulating the functionality of system by burning firmwares on chips making it harder to reverse engineer it.

In order to better understand embedded systems, we can look at differences from general purpose computers. General purpose computers are developed to run all types of general applications. However, embedded systems are designed for a fixed or a few number of dedicated functions. General purpose computers are reprogrammable by end users while embedded systems are not reprogrammable by end users. Embedded systems are smaller in size and run at fixed optimized speed for a specific purpose while general purpose computers are bigger in size with speed which does not need to be always predictable but faster is always better.

Embedded systems, once used only for single purpose time critical applications, are now becoming important for all devices used in our everyday life. Such devices are now able to run general purpose operating systems or application with little or no knowledge of hardware constraints. Although safety critical systems are far more restricted that so-called modern embedded systems, virtualization could bring advantages, by increasing their safety, reliability and security [21].

2.2.1 Virtualization on Embedded Systems

Adding a hypervisor to an embedded system adds flexibility and higher-level capabilities, morphing the embedded device into a new class of system [22]. Embedded devices are ubiquitous and are major part of our lives today. Their common use is in real time applications with hardware and software constraints. But why there is a trend seen today to use these devices in virtualized systems. One of the reasons is that users now want to run applications developed originally for general purpose OSes and still desire to achieve real time responsiveness. There is where virtualization comes in handy. With virtualization, we can enable concurrent execution of real time OS (RTOS) and application OS(Windows, Linux etc) on same hardware. Another benefit will be security which can be achieved by encapsulating vulnerable application OS in a separate VM thus preventing access to the rest of the system.

In a nutshell, there are many uses of deploying virtualization on embedded systems and with the development in multi-cores technology, we can expect innovative solutions in future embedded virtualized platforms.

2.2.2 ARM Embedded Platforms

Since the processor used in thesis is ARMv8 Cortex-A53, a brief introduction of ARM architecture in general and ARMv8 in specific along with its features will be provided in this section.

Introduction to ARM Architecture

ARM, originally **Acorn RISC Machine**, later **Advanced RISC Machine**, is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments [23]. ARM has the following RISC architecture features [24]:

- A uniform register file load/store architecture, where data processing operates only on register contents, not directly on memory contents.
- Simple addressing modes, where all load/store addresses are only determined from register contents and instruction fields.

With the increase in demands of new functionality and emerging market trends, ARM architecture has evolved over time. It has introduced the concept of **profiles** which define different versions of the architecture used for different types of processors which are aimed to to used in different market segments [24]. Table 2.1 shows these different profiles of ARM architecture. ARM processors are basically used to achieve high-performance at

Profile	Description
Architecture (‘A’) profile	Provides high performance and usually used in mobile and enterprise markets
Real-Time (‘R’) profile	Provides real time performance and used in embedded applications for automotive and industrial control.
Microcontroller (‘M’) profile	Provides time critical and real time performance for microcontroller market

Table 2.1: Description of ARM profiles

lower cost with efficient power consumption.

ARM processor modes and Registers

There are two categories of ARM processor modes i.e. privileged and non-privileged modes. Privileged mode is used to handle exceptions or to access system resources while non-privileged mode has restricted access to protected resources. Each processor mode uses a its own stack and a subset of registers. Table 2.2 shows different processor modes supported by ARM architecture. In all ARM processors, the following registers are available and accessible in any processor mode [1]:

Modes	Description	Category
User	Normal program execution	Privileged
Fast interrupt (FIQ)	Handles fast interrupts	Privileged
Interrupt (IRQ)	Handles regular interrupts	
Supervisor	Handles operating system functions. System enters into this mode when the power is applied.	
Abort	Handles Data Aborts and Prefetch Aborts and helps to implement virtual memory	
System	Handle operating systems function in user mode and uses same registers as User mode	
Undefined	Handles Undefined instructions with the support of software emulation of hardware co-processors	
Monitor	Provides support of switching between secure and non-secure states available on processors with security extensions	

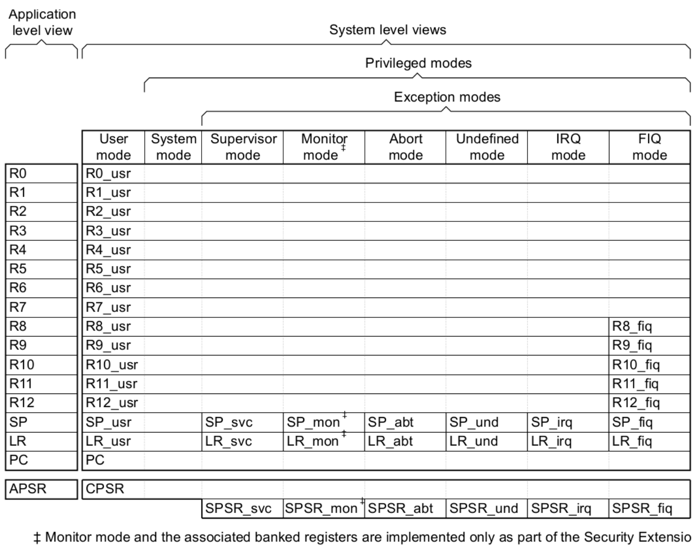
Table 2.2: Description of ARM processor modes

- 13 general-purpose registers R0-R12
- 1 Stack Pointer (SP)
- 1 Link Register (LR)
- 1 Program Counter (PC)
- 1 Application Program Status Register (APSR)

ARM processor has total of 37 registers (40 with security extension implementations) arranged in partially overlapping banks which help to context switch rapidly. Figure **??** shows the organization of general purpose registers of different ARM processor modes.

2.2.3 ARMv8-A Architecture

The basic feature of ARMv8 architecture is that it supports both 32-bit (AArch32) and 64-bit (AArch64) execution states.For AArch64 state, addresses are placed in 64-bit registers and instructions can use 64-bit registers for processing while on the other hand, AArch32 allows instructions to use 32-bit registers and addresses are placed in 32-bit registers [25]. For this thesis, AArch64 execution state has been used which supports up to four Exception levels, EL0 - EL3 and has 64-bit virtual addressing. Table 2.3 shows the description of exception model levels. The Cortex-A53 processor is a mid-range, low-power processor that implements the ARMv8-A architecture with Generic Interrupt Controller (GIC) v4 and ARM Generic Timer [26].



Application level view	System level views							
		Privileged modes						
			Exception modes					
	User mode	System mode	Supervisor mode	Monitor mode‡	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

‡ Monitor mode and the associated banked registers are implemented only as part of the Security Extensio

Figure 2.4: Organization of ARM registers for different processor modes. Taken from [1]

Exception Level	Description
EL0	Applications
EL1	OS kernel and associated privileged functions
EL2	Hypervisor
EL3	Secure Monitor

Table 2.3: Description of ARMv8 Exception Model Levels

ARMv8-A Memory Management

Memory memory unit (MMU) is a hardware that performs virtual address to physical address mapping. It does this by controlling table walk hardware which accesses translation tables held in main memory.Transalation tables hold virtual to physical address mappings and memory attributes which are then loaded into the Translation Lookaside Buffer (TLB) when a location is accessed [26]. With MMU enabled in system, applications can run independently in their own virtual address space without having the knowledge of physical addresses used by hardware. Figure 2.5 shows the MMU hardware in ARM architecture.

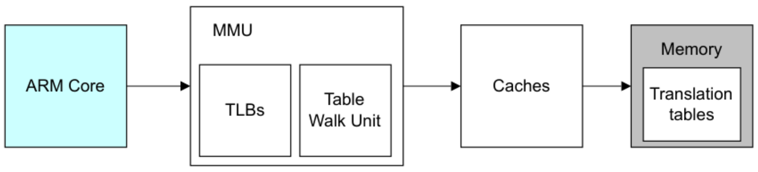


Figure 2.5: Organization of ARM registers for different processor modes. Taken from [2]

For ARMv8 hardware used in thesis, 48-bit virtual address with 4KB granule size has been used. Cortex-A53 processor uses a four level address lookup with 4KB page size. The 48-bit address has nine bits for each level of translation with the last 12 bits of original address defining the offset within the 4kB page size. Each level of translation lookup table has 512 entries. Bits 47:39 of the Virtual Address index into the 512 entry L0 table. Each of these table entries spans a 512 GB range and points to an L1 table. Within that 512 entry L1 table, bits 38:30 are used as index to select an entry and each entry points to either a 1GB block or an L2 table. Bits 29:21 index into a 512 entry L2 table and each entry points to a 2MB block or next table level. At the last level, bits 20:12 index into a 512 entry L2 table and each entry points to a 4kB block [3]. Figure 2.6 shows the division of 48 bit address for four levels of translation lookup used for 4KB granule size by MMU.

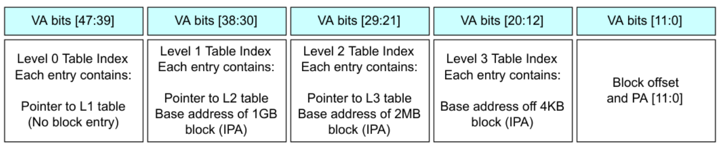

VA bits [47:39]
VA bits [38:30]
VA bits [29:21]
VA bits [20:12]
VA bits [11:0]
Level 0 Table Index Each entry contains: Pointer to L1 table (No block entry)
Level 1 Table Index Each entry contains: Pointer to L2 table Base address of 1GB block (IPA)
Level 2 Table Index Each entry contains: Pointer to L3 table Base address of 2MB block (IPA)
Level 3 Table Index Each entry contains: Base address off 4KB block (IPA)
Block offset and PA [11:0]

Figure 2.6: 48-bit address translation lookup for 4KB granule size on ARMv8 architecture. Taken from [3]

2.2.4 Device Emulation on ARMv8-A Embedded Systems

ARMv8-A architecture supports virtualization by implementing EL2 execution state for running hypervisor. Hypervisor in EL2 state is responsible for running multiple guests in non-secure EL1 state. Each guest can run applications in non-secure EL0 state. Address translation occurs in two stages in case of running virtualized guest operating systems. Stage 1 translation converts virtual addresses to intermediate physical address (IPA) which is managed by Guest OS in EL1 state. Stage 2 translation converts IPA to physical address which is managed by hypervisor in EL2 state. These IPA are treated as actual physical addresses by Guest OS. ARM virtualization extensions has made generic timers and the GIC interrupt controller virtualization aware. Hence CPU, memory, interrupts and timers can be emulated using full hardware virtualization. However, for I/O devices, para-virtualized drivers could be used.

On ARM architecture, device virtualization can be done using memory mapped devices. All reads/writes to devices gets trapped by hypervisor which should be capable of emulating device loads/stores. With ARM virtualization extensions, device virtualization has become more efficient. The main features of these extensions include introduction of a new higher privileged Hypervisor execution mode than Supervisor mode, mechanisms to aid interrupt handling and the provision of a System MMU to aid memory management, supporting: multiple translation contexts for multiple DMA capable masters, two levels of address translation and hardware acceleration and abstraction [27].
Currently, on ARM virtualized systems, I/O devices can be emulated using either Type-1 or Type-2 hypervisors. Type 2 hypervisor allows to reuse guest OS code especially of device drivers for different types of hardware. However,Type 1 hypervisors requires device drivers to be reimplemented for a wide range of hardware support. Xen, a Type 1 hypervisor on ARM, has avoided this issue by implementing a minimal amount of hardware support directly in hypervisor and allows a special privileged guest called Domain-0 to perform I/O using existing device drivers on behalf of other non-privileged guests [28].
In short, ARMv8 architecture provides efficient virtualization as follows:

- **CPU virtualization** with hypervisor in EL2 state configuring CPU to trap to sensitive and privileged instructions

- **Memory Virtualization** with hypervisor pointing to its own set of stage-2 translation tables for translating intermediate physical addresses to actual hardware addresses.

- **Interrupt Virtualization** with virtualization extensions support in ARM generic interrupt controller (GIC) to allow hypervisor to inject virtual interrupts to guests OSs which they can complete and acknowledge without being trapped in hypervisor.

- **Timer virtualization** by allowing guest VMs to configure virtual timer without trapping to hypervisor.

2.3 Overview of Phidias Hypervisor

PHIDIAS, the Provable Hypervisor with Integrated Deployment Information and Allocated Structures, is the statically configured hypervisor developed by Dr.-Ing. Jan Nordholz at TU Berlin Telekom Innovation Laboratories with the faculty of Security in Telecommunications [4]. It is second implementation of Principle of Staticity after Perikles which is being integrated in industrial automotive products at OpenSynergy [29].

2.3.1 Principle of Staticity

PHIDIAS is based on following Principle of Staticity which states that

Non-mandatory dynamic components of a hypervisor for an embedded system should be removed completely and if not possible, should reduce their dynamicity to generate a pure static and easily verifiable code by configuring the desired characteristics at compile time.

The basic idea behind developing such minimal hypervisor is to remove all dynamic elements from it in order to ease provability and reduce runtime complexity as well as memory footprint. For certification of software to be used in safety critical applications, static analysis is widely used. If the behavior of software is constrained to be as static as possible with limited or no dynamic elements, it can be reliably proved and thus used in safety critical real time applications.

2.3.2 Core Components of Minimal hypervisor implementation

Memory requirements of virtual machines are defined at compile time which results in requested memory allocations and alignment constraints to be verified and satisfied statically and assigning fixed physical addresses to each of those allocations. All desired page tables, list of mapped memory ranges available to software and memory areas are finally compiled into the resultant bootable image. Virtual CPU interface, simulating full privileged and unprivileged register banks, is provided to run unmodified guests with hypervisor responsible for trapping and emulating certain sensitive instructions. This hypervisor is instantiated on each physical CPU for a multicore architecture running guest VMs on individual cores. It's scheduler is based on a simple single-priority round-robin policy. Dispatching of interrupts to VMs is done by a implementing a static interrupt dispatch table in read-only memory. Passing through an interrupt to a VM is done by using a second read only dispatch table which determines the target VM for each interrupt line. PHIDIAS provides emulation of three basic hardware devices i.e. UART, timer, and interrupt controller. It does device emulation by implementing modifiable data structure to maintain the runtime state of emulated device and configuring guest physical memory ranges to which the device expects to respond to. Events and timers are implemented using event queue based on a single hardware timer with preallocation of timer events for signaling the end of time slice of virtual CPUs and timer events for each emulated timer device. Support of basic inter VM communication is also implemented using shared memory and signalling mechanism. This signalling mechanism is based on concept of capabilities. These capabilities are objects implemented internally in hypervisor that can be invoked to trigger desired actions e.g. triggering an interrupt to a target VM in case of inter VM communication. For systems without two stage address translation support in hardware, paravirtualized execution of virtual CPUs needs virtual translation lookaside buffer (VTLB).BVTLB implementation in PHIDIAS is based on two core components i.e. walker and pager. Walker component inspects effective guest page table in case of memory access fault and pager component adds hypervisor controlled two-stage translation of target address to the effective page table.

2.3.3 Basic Structure of PHIDIAS

The basic structure of PHIDIAS is shown in Figure 2.7.

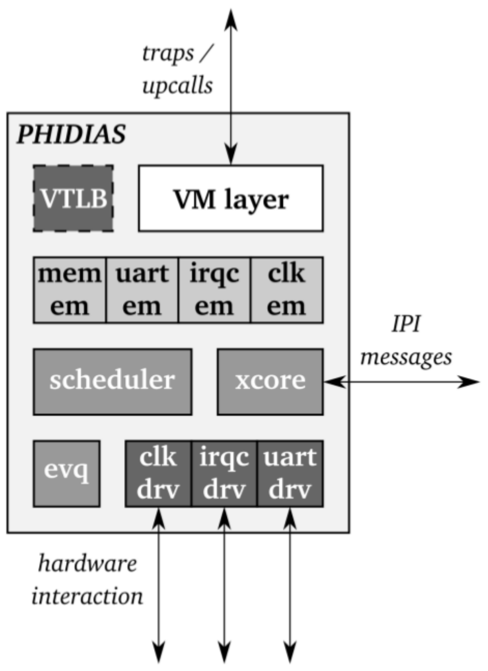


Figure 2.7: Phidias Structure. Taken from [4]

All mandatory components which are required for a hypervisor to function correctly are implemented in PHIDIAS. Following is a brief description of each of these basic components.

VM Layer

VM Layer is responsible for performing world switch between guests and hypervisor. It performs upcall to enter into guests world and dispatches events to appropriate components for handling them.

Emulation Layer

Emulation layer consists of minimal implementation of components necessary for hypervisor to operate. It includes UART, clock, Interrupt controller and a generic memory emulation. Generic memory emulation is added to run unmodified platform-specific Linux kernels without breaking their device specific functionality. Device driver sends memory requests to expected addresses but gets zero on reads and writes are discarded.

Scheduler and Xcore

At the heart of PHIDIAS, there is a scheduler with simple single-priority round-robin policy and Xcore components for relaying interprocessor interrupts between different instances of hypervisor and triggering interrupt capabilities.

Event Queue and Drivers for emulation

There is an event queue which is responsible for keeping track of programmed timer events and emulation drivers for minimal functionality is also present.

2.3.4 Static Configuration and Final Image

PHIDIAS is a static hypervisor which is built using a modifiable scenario specific configuration through an XML based compile time utility called Schism, the Static Configurator for Hypervisor-Integrated Scenario Metadata. There are two types of configuration elements i.e. Hypervisor configuration and VM specific configuration elements.

Hypervisor Configuration Elements

- Selection of CPU architecture and target platform SoC
- Physical and virtual Hypervisor base load address
- Selection of drivers for hardware devices and for the required emulation devices

VM specific Configuration Elements

- Number of virtual CPUs per guest
- memory configuration per guest
- List of capabilities of each VM

- Assignments of pass-through interrupts to VMs
- Types, parameters, and corresponding emulated memory for selected emulated devices

3 Overview of Xen

This section provides background information on the Xen hypervisor and core components of its split device driver architecture for I/O emulation.

3.1 Introduction of Xen

The Xen Project hypervisor is an open-source type-1 or baremetal hypervisor, which makes it possible to run many instances of an operating system or indeed different operating systems in parallel on a single machine (or host) [30]. It is one of the popular open-source hypervisors which can provide both para-virtualization and full virtualization solutions.In fact, it is the only available type-1 hypervisor which is open-source. It has been used in server and desktop virtualization and fueling the biggest clouds and web services in production today e.g. Amazon Web Services.

Xen was developed at University of Cambridge in 2003 by Ian Pratt, a senior lecturer in the Computer Laboratory, and his PhD student Keir Fraser [5]. It was acquired by Citrix in 2007. Since April 15, 2013, Xen Project has become a Linux Foundation Collaborative Project with the following companies contributing to and guiding the Xen Project as founding members are: Amazon Web Services, AMD, Bromium, Calxeda, CA Technologies, Cisco, Citrix, Google, Intel, Oracle, Samsung and Verizon [31].

The basic components which work together to provide virtualization solution include Xen Hypervisor, Privileged guests called **Domain-0 or Dom0** and unprivileged guests called **Domain-U or DomU**. Xen hypervisor is a small software which directly runs on hardware and is responsible for CPU scheduling ,memory management and interrupt handling. After Xen hypervisor is booted, it launches Domain-0 guest which has direct access to all hardware and has native device drivers for performing I/O operations. Other unprivileged guests or domains access hardware and perform I/O via Dom0. Dom0 is also responsible for launching and managing DomUs with the help of control stack called Toolstack running on it. Figure 3.1 shows the architecture of Xen virtual environment.

3.1.1 Guest Virtualization Types in Xen

Xen provides two types of virtualization modes for guests:

- **Paravirtualization (PV)**, first introduced by Xen, is virtualization technique in

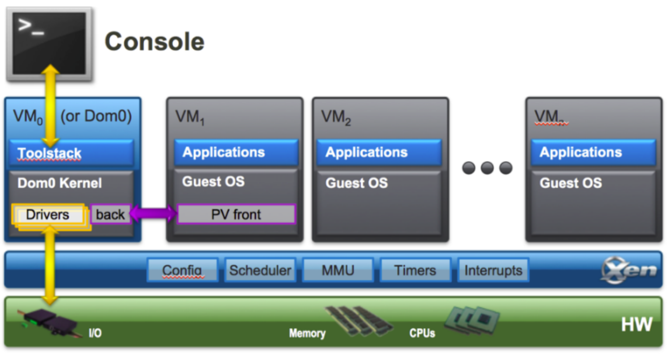


Figure 3.1: Xen Architecture Taken from [5]

which guests are modified and are aware of being run on a hypervisor. A PV guest requires PV enabled kernel and PV drivers to be present to run without emulation. Dom0 in Xen is a privileged PV guest which has access to actual device drivers and hardware in the system and provides an interface to control and manage other VMs.

- **Hardware-assisted or Full Virtualization (HVM)** is a virtualization mode which requires virtualization hardware extensions (Intel VT or AMD-V hardware extensions) to be present on host CPU. Guests kernels are used unmodified and hence Windows can run as HVM guest on Xen.Xen uses Qemu on HVM guests for hardware emulation and hence are slower than PV guests. However, paravirtualized drivers can be used for I/O on HVM guests to increase performance of system.

Guests with both types of virtualization can be run at the same time on Xen. It is also possible to use paravirtualization on HVM guests and vice versa. This mixing of modes has introduced two more modes of virtualization on Xen which are PVHVM and PVH. In PVHVM guests, optimized paravirtualized drivers are used for disk and network virtualization on hosts with hardware virtualization extensions enabled. PVH are basically paravirtualized guests which use PV drivers for I/O operations and use hardware virtualization extensions for others.

There are pros and cons of each mode of virtualization in Xen. Full virtualization provides full emulation of underlying hardware to guests which require no modifications in their OSes. However, performance is degraded while providing full emulation of entire system by VMM. Also HVM guests use trap and emulate model for execution of sensitive and privileged instructions which can cost to hundreds to thousands of cycles [32].. On the other hand, paravirtualization allows modified guests to run on a system with an abstraction of similar physical hardware on system and hence provides near native performance as compared to HVM guests. It replaces the sensitive instructions with hypercalls to VMM and allows combining several hypercalls into one hypercall to reduce transitions between Guests OS and VMM [32].

3.2 Device I/O Emulation on Xen

Xen hypervisor virtualizes CPU, memory, interrupts and timer. It does not have any knowledge of I/O devices. Access to actual physical I/O devices and their native device drivers is present in privileged Domain-0. Unprivileged guests can get access to I/O devices through Dom0. Xen assigns all I/O devices to Dom0 which is then responsible for MMIO remapping and interrupt handling.

Device virtualization in Xen is done using a pair of paravirtualized drivers called **frontend and backend drivers**. For each class of hardware devices i.e. disk, network, console, framebuffer, mouse, keyboard, etc, a pair of paravirtualized drivers is present in system. Paravirtualized backends are implemented in Dom0 with their corresponding frontends being implemented in DomU. These paravirtualized drivers are implemented as kernel drivers. However, some backends can run in QEMU in userspace. Communication between frontends and backends is done through a shared memory ring protocol and events notifications mechanism provided by Xen. Xen provides tools to setup communication framework between frontend and backend drivers in Dom0 [6]. Figure **??** shows the I/O device virtualization architecture in Xen.

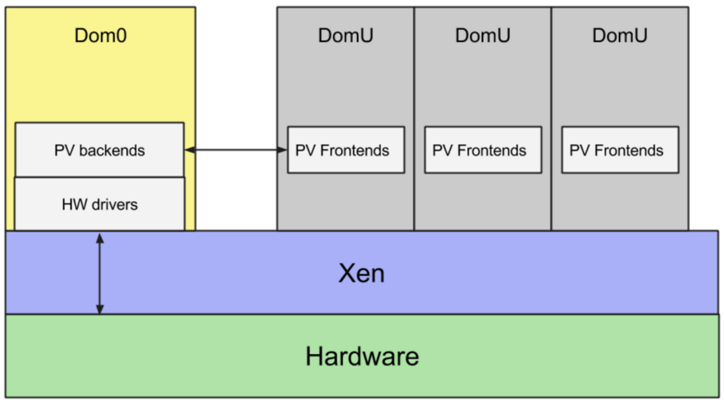


Figure 3.2: Xen I/O Device Virtualization Architecture. Taken from [6]

In order to provide isolation, security and disaggregation, Xen has introduced the concept of driver domains. Such domains are unprivileged guests running backends and native drivers for I/O device virtualization. If such a domain is compromised or crashed, it will not affect other guests or domains. Figure 3.3 shows the architecture of driver domains in Xen.

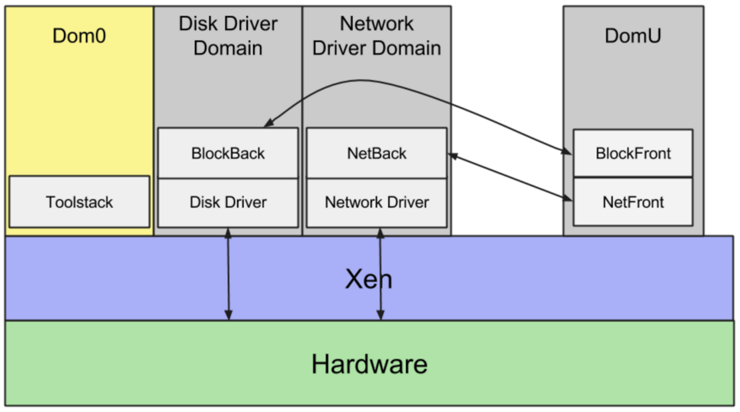


Figure 3.3: Xen Driver Domains Architecture. Taken from [6]

3.2.1 Xen on ARM

Xen on ARM is a simple, smaller and faster as compared to its x86 counterpart. The main features of Xen on ARM are described as follows:

- It does not perform any emulation. There is no QEMU emulation in Xen on ARM.
- It exploits hardware virtualization support for memory management, interrupts and timer virtualization.
- It uses paravirtualized pair of drivers for I/O virtualization.
- It runs entirely in hypervisor mode and provides an HVV hypercall to kernels to switch between hypervisor mode and kernel mode.
- It uses virtualization aware Generic Interrupt Controller for interrupt handling.
- It uses virtualization aware Generic Timers to provide timer virtualization to guests.
- It supports one type of guests which exploits hardware virtualization as much as possible and uses paravirtualized interfaces for I/O device virtualization.

Figure 3.4 shows the simple architecture for Xen on ARM platforms.

3.3 Xen Split I/O Driver Model

With the widespread use of embedded systems in multiple applications, diversity in I/O devices used by such systems has increased significantly. Supporting a large of I/O devices in Xen would make things worse for maintaining Xen hypervisor code and avoiding bugs. Most operating systems e.g Linux provide support for a large number of devices and reusing this capability would make a cleaner, smaller and much simpler

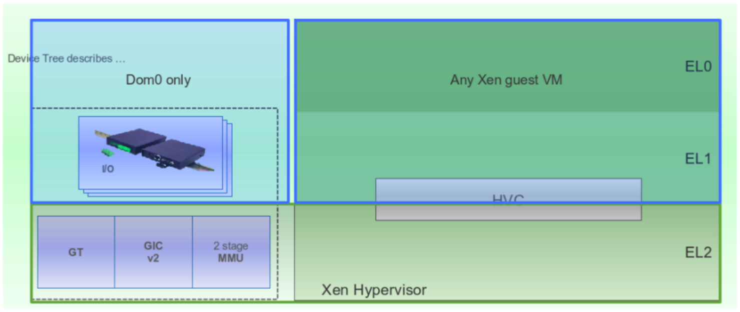


Figure 3.4: Xen on ARM Architecture. Taken from [6]

hypervisor. Xen has delegated all I/O hardware devices to privileged Domain-0 or special unprivileged driver domains. All other guests access I/O devices through Dom0 or driver domains. Xen provides mechanisms for communication between guests which use split driver model for multiplexing and using hardware I/O devices. Xen support Net, Block, console, keyboard, mouse, framebuffer, XenGT(intel Graphic card), 9pfs, PVCalls, Multi Touch, Sound and Display devices [33]. In the next sections, details of basic components of split driver model of Xen will be explained.

3.3.1 Basic Components of Xen Split I/O Driver Model

Xen device drivers typically consists of four major components:

- The native I/O device driver
- Top half or Frontend of the split driver
- Bottom half or backend of the split driver
- Shared ring buffers

The real drivers for I/O devices are present in Dom0 or driver domains for accessing actual hardware. They are interfaced with backend drivers of the split driver which provides a generic interface and I/O device multiplexing functionality. Frontend driver is present in unprivileged guests and communicate with backends using shared memory ring buffers. Frontends write requests on these buffers and backends write responses which are signaled through xen event notification mechanism. Successful shared memory communication between frontend and backend requires xen grant tables, event channels, xen bus and Xenstore to be working in system. These mechanism will be explained in the later sections. Figure 3.5 shows the basic architecture of Xen split driver mode.

3.3.2 Xen mechansims used by Split I/O Driver Model

There are basically four mechanisms provided by Xen which are used by Split Driver model of Xen to work successfully. These are as follows:

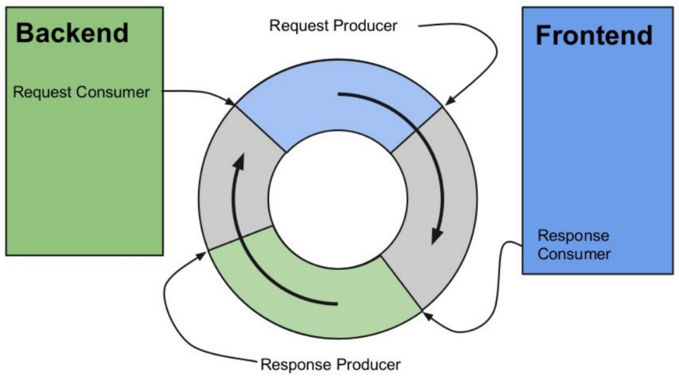


Figure 3.5: Basic Xen Split Driver Architecture. Taken from [7]

- Grant Tables
- Event Channels
- Xenstore
- XenBus Protocol

The split drivers across domains use these mechanisms to communicate and access hardware devices. Following sections provide brief explanation of each of these mechanisms.

Xen Grant Tables

Grant tables is a mechanism to provide shared memory to guests. Each guests can manipulate the shared memory on page level granularity. Grant tables provides two types of memory sharing operations to guests i.e. memory sharing and memory transferring. Since all physical memory is mapped by Xen hypervisor, copying the data between domains is done by hypervisor without modifying page tables. Transferring a page from one domain to another is also available which changes the page owner and modifies the page tables accordingly. To identify the page being shared or transferred, guests uses grant references which are integers indexed into an array of grant entries in grant table. These grant references are placed in Xenstore ,a virtual file system for device discovery in Xen, to communicate shared page information to other domains.

The interface to grant tables is provides by Xen in the form of hypercalls for grant table operations. Two basic operations that can be performed on grant tables are mapping and transferring pages. Mapping a page removes original reference of page in sender domain's address space while transferring causes the page to leave calling domain's address space. Mapping is used by drivers to implement interdomain communication using shared memory while transferring is used to move data between domains.

Xen hypervisor creates four types of structures to implement grant table mechanism:

- Shared Grant table is created and shared by Xen for each guest. Guest writes into entries in table and Xen performs the desired operation specified by hypercalls. Four pages are allocated and shared with each guest during initialization of each domain. A maximum of 32 pages can be allocated for shared grant tables.
- Active Grant table is created and maintained by Xen to keep track of active grants per domain. Four pages are allocated initially for implementing active grant table per domain by Xen.
- Mapped track table is created and maintained by Xen per domain for each mapped page. Initially 1024 map track entries are allocated for each domain by Xen.
- Status Grant table is created and maintained by Xen for keeping track of status of each grant per domain.

Figure 3.6 shows the basic structure of shared grant table.

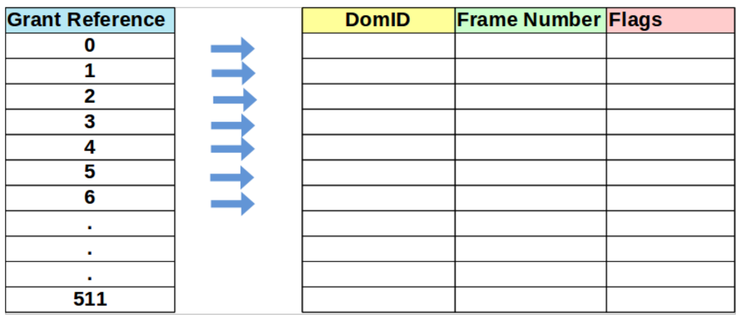


Figure 3.6: Basic structure of shared Grant table

Grant Table Use for Shared Ring Buffers Device I/O rings which are used for communication between split drivers are built on top of shared memory provided by grant table mechanism. Frontend drivers creates ring buffer pages and grant access to backend. Backend drivers map these shared pages for ring buffers and communication channel is established across domains. Figure 3.7 shows the actions performed by split drivers on ring buffers for interdomain communication.

Xen Event Channels

Event channels is a mechanism for asynchronous delivery of notifications between domains. These are used with shared memory mechanism to provide message passing between split drivers in domains. Events are similar to Unix signals. Each event represents one bit of information about which event has occured. There are four types of event channels supported by Xen:

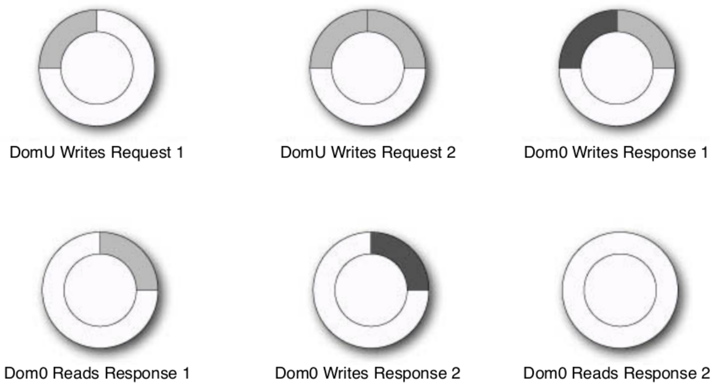


Figure 3.7: Request/Response sequence on ring buffers. Taken from [8]

- Interdomain events are used by split drivers for notifying each other about data waiting to be transported to other domain. These are bi-directional.
- Physical IRQ are used to bind actual hardware IRQs to event channels. These are used by Domain-0 or driver domains to access various devices under their control by mapping physical IRQs to event channels.
- virtual IRQs (VIRQ) are used to bind IRQs of virtual devices e.g virtual timer or emergency console to event channels.
- Intradomain events are used to send events between virtual CPUs of a single domain similar to interprocessor interrupt (IPI) mechanism. These are bi-directional.

Event channel creation is a two-stage process. In the first step, an event source is bound to event channel and in second step, an event handler is registered for handling triggered event. Event channel is an abstraction of sending asynchronous notifications between domains. Each channel has two endpoints which are called ports in local domain. After binding an event channel to remote port, a domain can send an event to local port via hypercall through Xen hypervisor. Xen is then responsible for finding the remote end of channel and remote domain and route events to destination.

Each virtual CPU in the guest on Xen has an event channel bitmap associated with it. This event channel bitmap is shared with Xen in shared info page created during initialization of guest. When an event is received by Xen for a particular guest, it sets event channel upcall pending flag, desired bit of event and corresponding word which contains set event bit. All these fields are present in shared info page. There are also mask bits for disabling event delivery to guests. Events delivery can be masked both by individual event type or disabling/enabling all together.

Two implementations of event channels are supported in Xen [34]:

- 2-level event channel ABI is de-factor implementation for event channel mechanism for which 32 bit domain supports up to 1024 event channels and 64 bit domain supports up to 4096 channels.A 2-level search path is used for finding the set event bit in event channel bitmap. For the thesis, 2-level event channel has used since it meets the requirements for porting I/O split driver to PHIDIAS.
- FIFO-based event channel ABI has lockless queues for event queues with configurable number of event channels and event priorities. It can support more thanm 100,000 event channels, with scope for 16 different event priorities.

Figure 3.8 shows the basic architecture of 2-level event channel implementation.

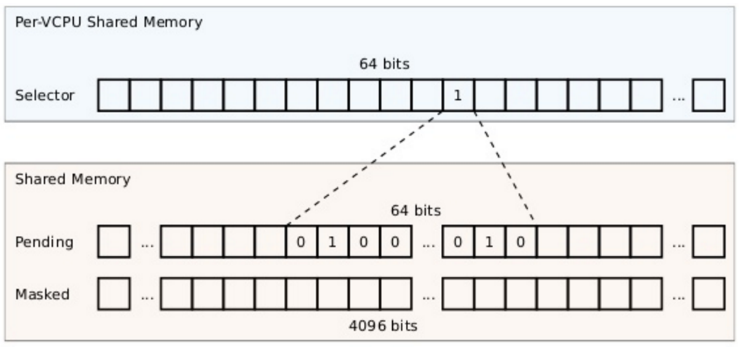


Figure 3.8: 2-level event channel implementation in Xen. Taken from [9]

Xenstore

Xenstore is a hierarchical storage system maintained by Dom0 and accessed by guests through shared memory page and event channels. It is a tree database (tdb) of storage and configuration information located in Dom0. Xenstore has no hypercalls and hence can be compiled and run without depending upon Xen hypervisor. However, it needs uevent channels to communicate with domains using ring buffers built on top of shared memory pages. Figure 3.9 shows the basic hierarchy of xenstore filesystem.

There are two rings created on a shared memory page per domain. One for transmitting requests and other for reading responses asynchronously. Guests write requests on ring buffer and xenstore writes responses. Only Dom0 has permissions to write or modify data stored in database of xenstore. Other guests. If DomU wants to write data in xenstore filesystem, correct permissions must be granted to it by Dom0. Each guest shares its own memory page with Xenstore and creates an event channel to communicate with it.Xen provides XL tool [35] to create unprivileged domains by Dom0. This tool is also responsible for registering unprivileged domains with Xenstore and map their shared memory pages into Xenstore userspace.

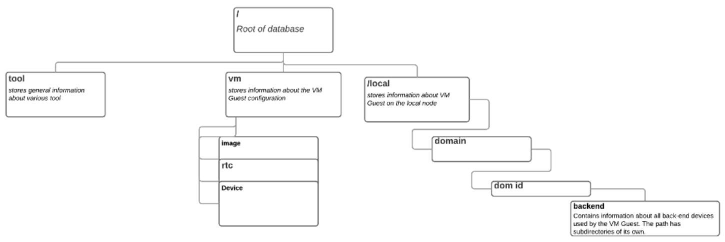

/
Root of database
tool
stores general information about various tool
vm
stores information about the VM Guest configuration
/local
stores information about VM Guest on the local node
image
rtc
Device
domain
dom id
backend
Contains information about all back-end devices used by the VM Guest. The path has subdirectories of its own.

Figure 3.9: Basic hierarchy of xenstore filesystem

Xenstore is running as a xenstore daemon called Xenstored and allows dom0 and guests to access information about configuration and status of the system [36]. Xenstore filesystem stores information in the form of directories which contains other directories or keys. Hence its structure is similar to a dictionary with key and value pairs for information. It supports handling multiple requests within a single transaction atomically to provide consistent view of stored information. The design of xenstore interface is based on central polling loop which reads requests, polls watches and invokes callbacks [37].

XenBus

Xenbus [38] is a protocol built on top of the XenStore used for enumerating and connecting split device drivers in Xen. Xenbus provides a bus abstraction to paravirtualized drivers in Xen to connect domains with each other. It requires grant tables, event channels and xenstore to function correctly. XenBus implements a state machine which keeps track of several states of virtual drivers during the process of enumeration of virtual devices. Each split driver registers itself with XenBus which in turn calls the corresponding probe function. Frontend drivers can only be probed once corresponding backend drivers and xenstore are up and running.

Both halves of Xen's split driver model should implement the core component of Xenbus interface which is the xenbus state enumerated type. This means that while registering with XenBus, paravirtualized drivers should provide with their own implementation of ***otherend_changed*** function which switches states of the driver and performs necessary setup of virtuals device depending upon the changes in state of other end of driver. Xenbus interface provides functions to split drivers to register watchers with Xenstore filesystem which gets invoked upon the changes in watched node's data. Figure 3.10 shows states of backend and frontend drivers managed by Xenbus protocol.

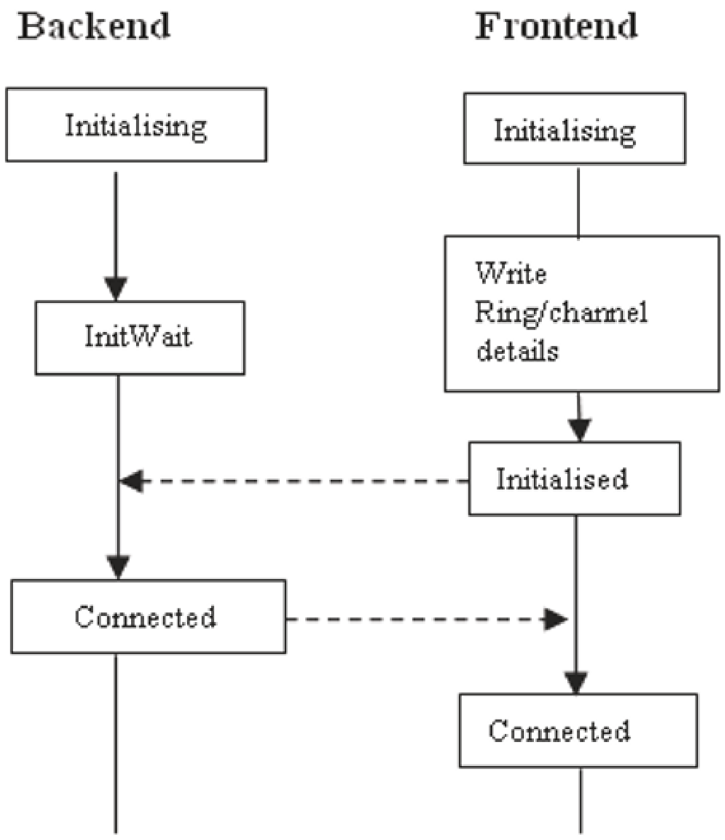


Figure 3.10: States transition of frontend and backend drivers implemented by XenBus. Taken from [10]

4 Porting Environment and Requirements

In this chapter, the basic requirements and environmental setup for porting Xen split driver model to PHIDIAS will be discussed.

4.1 Environmental Setup for Porting

In this section, environment used for porting I/O device emulation of Xen to PHIDIAS will be explained. Basically,the environment consisted of an ARM target platform and a Linux PC with required tools and source code. The details of target platform and source code used are given below.

4.1.1 Target Platform

HiKey (LeMaker version) [39] was used as a target platform which has the following features:

- Kirin 620 SoC
- ARM Cortex-A53 Octa-core 64-bit up to 1.2GHz (ARM v8 instruction set)
- 2GB LPDDR3 DRAM
- On-board 8GB eMMC nand flash storage
- TI WL1835MOD 2.4 GHz Wireless card

4.1.2 Linux kernel version

The Linux kernel used for running Xen on HiKey was 4.1.15 obtained from [40]. Linux kernel used for running PHIDIAS was linux-4.11-rc2 obtained from [41].

4.1.3 Xen version

Xen 4.7.0 was used for setting up reference framework for porting taken from [42].

4.1.4 PHIDIAS version

Table 4.1 describes commit ids for components of PHIDIAS and their git repositories used in current project.

PHIDIAS component	commit id	git repo
xml	30d880a5b2b28c7032c dce4b564496ab08f06a15	git@gitlab.sec.t-labs.tu-berlin.de:phidias /xml.git
core	3a1771b3aa6cf2b789805 d849e74e8ffbdc9dbc3	git@gitlab.sec.t-labs.tu-berlin.de:phidias /serial_multiplexer.git
abi	8d8c3d1251799701eb 82e5a54fb7ec8075a30ce	git@gitlab.sec.t-labs.tu-berlin.de:phidias/abi.git
serial_multiplexer	968b913fa865e7129a9 10c79818153b361cbaf6e	git@gitlab.sec.t-labs.tu-berlin.de:phidias /serial_multiplexer.git

Table 4.1: Description of PHIDIAS components commit ids and git repos used for porting

4.2 Requirements for Porting

Following requirements were considered and fulfilled while porting Xen I/O split driver model to PHIDIAS:

- Keep the changes as minimum as possible in PHIDIAS hypervisor code. For the current project, no additional changes are added in PHIDIAS besides adding serial multiplexer functionality for getting serial output from guests running on different physical CPUs.
- Keep the changes as minimum as possible in Xen domains Linux source code and make changes in separate generic interfaces exposed to all xen's split drivers. This also has been satisfied in current work by adding changes usable by split drivers for all types of I/O devices.
- Add required changes in Linux kernel code cleanly easy to maintain future releases. Two major changes were added for porting work. One was to create a separate memory ZONE name ZONE_XEN in Linux kernel used by split drivers for allocating pages for sharing and another was to register a custom IPI handler for software generated interrupts.
- Port Xen's frontend and backend of network virtual device to PHIDIAS for the proof of concept of flexibility of the static hypervisor and adding networking support between guests.
- Tools (iperf and ping) which are used for testing should be same. Ping utility used is of BusyBox v1.22.1 and iperf is of version 2.0.10 (11 Aug 2017) pthreads.

- Keep changes small in Xenstore userspace tool. For current work, only one change is added to introduce domU with Xenstore manually. In original Xen setup, this is done via XL tool as explained previously 3.3.2 which depends upon Xen hypervisor hence not used in our porting setup.
- At least two guests should be able to communicate via ported split driver.

5 Design and Implementation of Porting Basic Components of Xen Split Driver Model

For porting Xen's split driver model, three major components i.e. Grant tables, Event channels and Xenstore would be modified to use static memory and PHIDIAS xcore and capability feature. In this section, design and implementation of applying ***Principle of Staticity*** of PHIDIAS for xen I/O drivers will be discussed.

5.1 Porting Xen I/O Emualation Layer to PHIDIAS

In this section, design and implementation of necessary components of split driver model of Xen which were ported to static hypervisor are discussed.

5.1.1 Porting Shared Information Pages

Shared info page is used to share virtual machine state with Xen hypervisor. It includes information about virtual CPU (vCPU) state, event channels and wall clock time information. Each guest allocates a zerod page for shared info page from kernel and registers it with Xen hypervisor through a hypercall. In our case, a static memory range has been configured to allocate shared info pages. These pages are shared between guests instead with the PHIDIAS hypervisor. The size of this static memory is dependent upon the total number of configured guests in system. Each guest obtains its share info page from this static contiguous memory range using its domain ID defined in Linux configuration file (CONFIG_XEN_DOM_ID). Dom0 with id 0 initializes the entire range with zeros. The shared info page is mapped in linux guest as un-cached so that other guests would get the consistent view of each other's shared information. Figure 5.1 shows the approach of implementing shared info pages for two guests in PHIDIAS.

5.2 Design of Porting Grant Tables

As described in 3.3.2, grant tables are a mechanism for sharing memory pages across domains. Each guest allocates pages for grant tables and shares them with Xen hypervisor through a hypercall. By design, Xen does not support swapping which means that even unused part of allocated memory to a guest will be not used by other guests. To solve this problem, Xen uses **ballooning**. Ballooning is way of dynamically increasing

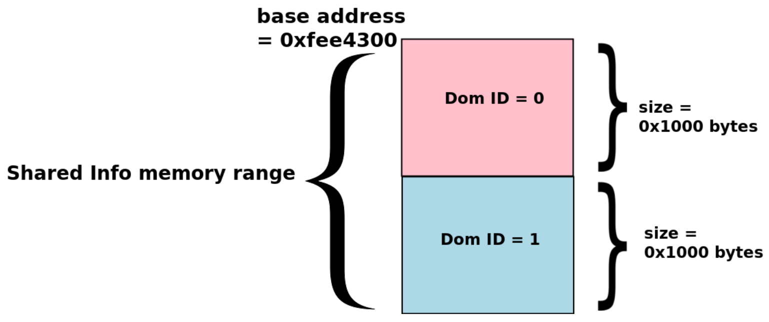


Figure 5.1: Basic approach of implementation of shared info pages in PHIDIAS for two guests

or decreasing size of allocated memory from hypervisor's memory pool. Memory visible to each guest can be configured in Xen at boot time. For Dom0, it could be specified in grub configuration file and for domU, it is specified in XL tool's guest configuration file. If the guest uses less memory than configured amount, it can return nused blocks of memory to hypervisor. However, guest can not retrieve more memory through ballooning than its maximum configured amount of memory. Memory allocated to unprivileged guests is taken from Dom0 memory pool and hence it Dom0 memory balloon's down every time a new guest is started.

For the current work, Xen balloon driver has been disabled in Linux guest kernel using configuration option CONFIG_XEN_BALLOON. In the native code of xen domains, ballooning is used to allocate pages for grant tables, XenBus ring buffers and netback driver for mapping packets received from netfront. In our ported setup, it has been replaced by defining two globally shared readable/writable memory ranges statically in PHIDIAS configuration option of guests. One memory range is defined to allocate static memory for grant table frames while the other is defined to be used by all split drivers for establishing communication for I/O virtualization. Required pages are allocated from these two memory ranges and mapped un-cached in address spaces of guests.

5.2.1 Static Memory for Grant Frames

In Xen, maximum number of grant frames is defined to be 32. For the current thesis, since two guests were used for testing, a total of 64 contiguous pages had been configured for grant tables usage. Each guest uses 32 pages for its own grant table implementation and accesses other guests' grant table by mapping these into its address space un-cached. For performing operations on grant tables, each guest gets the reference of unused entry from grant table and updates it with its domain ID, shared frame number and desired flags. The guest then sends the obtained grant reference through ring buffers to the other guest. In native Xen domains, hypercalls are used to perform grant operations i.e. granting access to remote domain, mapping, transferring and copying pages across domains. In ported static setup, guest had access to remote's domains grant tables by mapping those directly into its address space. Figure 5.2 shows the configuration of grant table static memory range for two guests in PHIDIAS.

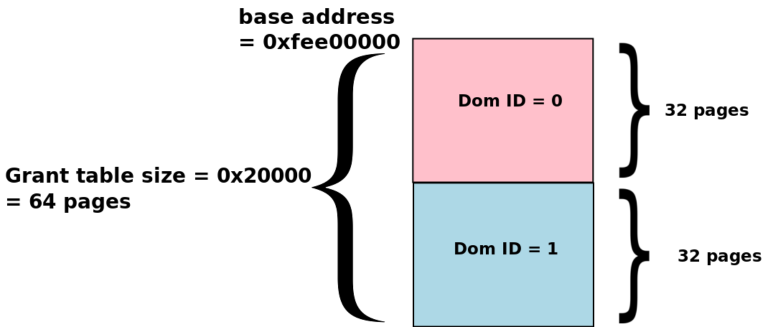


Figure 5.2: Static memory configuration for grant table in PHIDIAS for two guests

5.2.2 Static Memory for I/O Split Drivers Usage

I/O Split drivers of Linux guests allocates pages using kernel APIs for the allocation of page frames and share with other guests via grant table hypercalls. In order to use the same Linux page allocation APIs and keep changes in split drivers as minimum as possible, a new memory zone named **ZONE_XEN** had been added in Linux kernel. A static globally shared memory of size 8192KB has been allocated in PHIDIAS from which each guest can allocate its respective zone. The size of zone for each guest was set to 4096 KB i.e. 1024 pages. Each guest obtained the starting address of its zone using base address configured in PHIDIAS for zone memory range and its domain ID as shown in listing 5.1.

```
xen_zone_size = 0x400000;
xen_zone_start_addr = 0xfef00000 + (CONFIG_XEN_DOM_ID * xen_zone_size);
```

Listing 5.1: Code snippet for calculating start address of guest ZONE_XEN

Figure 5.3 shows the memory configuration for areas of ZONE_XEN for two guests in PHIDIAS.

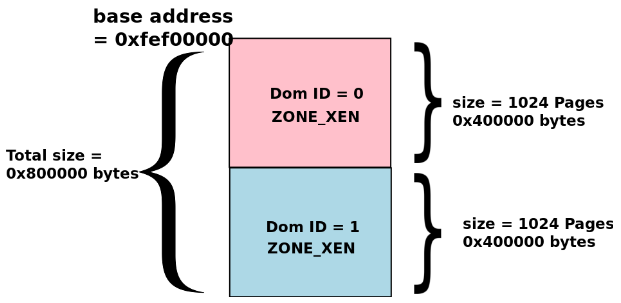


Figure 5.3: Static memory configuration for ZONE_XEN in PHIDIAS for two guests

Each guest knows about the address range of other guests ZONE_XEN. In order to access remote guests pages allocated from their ZONE_XEN, each guest maps the areas of other guests zones into its address space during its initialization and creates a hash table of virtual addresses of mapped pages indexed with physical page frame numbers. This hash table was implemented for ported split drivers for two reasons:

- To speed up the process of finding mapped virtual address of remote's guest shared page using the physical frame numbers obtained from remote guest's grant table which are already mapped as explained in 5.2.1.
- Function ioremap can not be called in interrupt context which is the case of mapping shared pages through grant tables in split drivers for most of the scenarios.

Figure 5.4 shows the basic structure of hash table for mapping remote guest's physical frame numbers of shared pages to mapped virtual addresses in local address space.

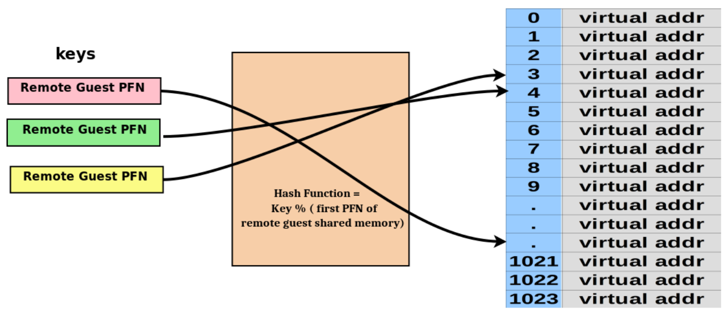


Figure 5.4: Structure of hash table used for accessing remote guest shared pages of zone xen

5.3 Design of Porting Event Channels

As explained in section 3.3.2, event channels are used to send asynchronous notifications among guests in Xen. On ARM, software generate interrupts (SGI) are used for inter-processor interrupts. There are 16 SGI available on ARM architecture. In Xen guests, there is a common interrupt handler **xen_arm_callback** for handling notifications from remote guests via event channels. Xen guest uses PPI 31 for event IRQ and has a an interrupt property in its hypervisor node of flattened device tree provided to it by hypervisor as shown in listing 5.2.

```
hypervisor {
    compatible = "xen,xen", "xen,xen-4.7"; moredelim//moredelimversion
        moredelim moredelimofmoredelim moredelimthemoredelim moredelimXenmoredelim
        moredelimABI
    reg = <0xb0000000 0x20000>; moredelim               moredelim//moredelim
        moredelimGrantmoredelim moredelimtablemoredelim moredelimmemorymoredelim
        moredelimarea
    interrupts = <1 15 0xf08>; moredelim               moredelim//moredelimevent
        moredelim moredelimnotificationsmoredelim moredelimIRQ
};
```

Listing 5.2: Xen Hypervisor node in hi6220 flattened device tree

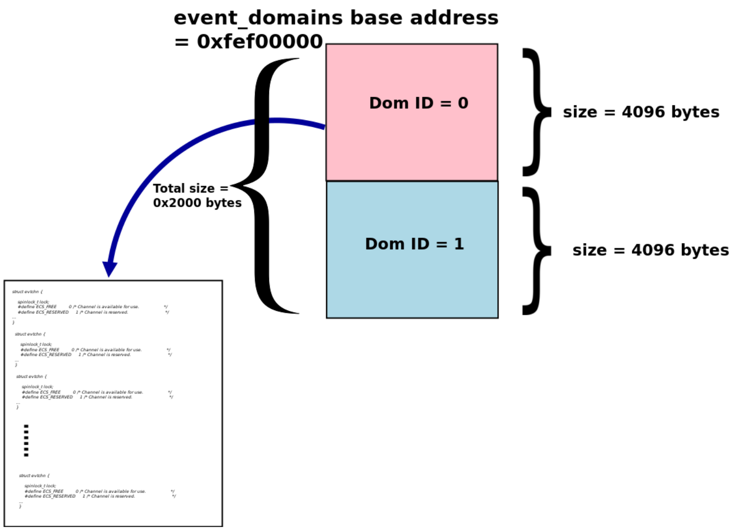
In PHIDIAS, SGIs are used to send interprocess interrupts using its **xcore mechanism**. First 6 SGI interrupts are used by Linux kernel. Our port could use one from the remaining SGIs for registering **xen_arm_callback** handler for Xen event IRQ. However, in Linux kernel version 4.11_rc2 used in current work, handle_IPI function used for handling SGIs only handles first 6 SGIs already registered by Linux kernel. For remaining SGIs, it shows a default warning of *Unknown IP*. To handle this, some modifications were made in IPI handling code of Linux kernel. For SGIs greater than 5, modified IPI handling function checked whether some handler is registered for incoming interrupt and then called the registered handler if found any. A kernel level API for registering IPI handler for SGIs was added. Web source [43] has been used as a reference for implementing generic API for registering custom IPI handlers.

SGI 9 had been used for Xen events in PHIDIAS for interdomain communication. 2-level event channel support had been ported which used two-level bitmap to speed searching. The first level is a bitset of words which contain pending event bits. The second level is a bitset of pending events themselves. All hypercalls in Xen guests were replaced with local guest's kernel code working on shared memory and IPIs.

5.3.1 Static Memory Allocation for Event Domains Pages

Xen hypervisor maintains a structure for each event per domain which stores necessary information e.g. VCPU for local delivery notification, event channel type, port number and priority etc. When a guest issues hypercall to send an event to remote guest, Xen hypervisor uses this structure to find remote domain ID and remote port and injects interrupt into destination guest.

In current work, all maintenance of event domain structures and triggering of IPI had been moved into Xen's guest domain. For remote sharing of event channel structures of guests, a static globally shared memory named **event_domains** has been allocated and mapped to guests' domains. In our implementation, the number of event channels each guest could support was limited to the amount of event domain structures that a single page could hold. Each guest had access to remote guest's event domain page containing corresponding event domain structures and could write its domain id and local event port into them while binding interdomain event channels. Figure 5.5 shows the basic structure of sharing event domains information in PHIDIAS for two guests.



```
struct evtchn {

   spinlock_t lock;
   #define ECS_FREE        0 /* Channel is available for use.            */
   #define ECS_RESERVED    1 /* Channel is reserved.                     */
...
}

struct evtchn {

   spinlock_t lock;
   #define ECS_FREE        0 /* Channel is available for use.            */
   #define ECS_RESERVED    1 /* Channel is reserved.                     */
...
}

struct evtchn {

   spinlock_t lock;
   #define ECS_FREE        0 /* Channel is available for use.            */
   #define ECS_RESERVED    1 /* Channel is reserved.                     */
...
}

struct evtchn {

   spinlock_t lock;
   #define ECS_FREE        0 /* Channel is available for use.            */
   #define ECS_RESERVED    1 /* Channel is reserved.                     */
...
}
```

Figure 5.5: Basic structure of sharing event domains information in PHIDIAS for two guests

5.3.2 Adding Capabilities for IPI in PHIDIAS

In case of split driver model of Xen, guests use event channels in two scenarios:

- Sending event notifications between Xenstore userspace application and guest's paravirtualized I/O driver behaving as intradomain events.
- Sending event notification between two frontend and backend of split drivers behaving as interdomain events.

For the above two types of events, two capabilities had been added in guest configuration on PHIDIAS as shown in listing 5.3.

```
For Dom0 Linux guest 1
      <cap type="ipc" target_xref="linux2" param="0x9" />
      <cap type="ipc" target_xref="linux1" param="0x9" />
For DomU Linux guest 2
      <cap type="ipc" target_xref="linux1" param="0x9" />
      <cap type="ipc" target_xref="linux2" param="0x9" />
```

Listing 5.3: Capabilities added for event notifications in guest configuration on PHIDIAS

Both capabilities had type **ipc**. First capability had index 0 with destination selected to be remote guest and second capability had index 1 with destination chosen to be itself. First capability was used for **interdomain events** and second capability emulated

intradomain events. Both these capabilities triggers SGI 9 for xen events in linux guests.

5.4 Design of Porting Xenstore

As explained previously in section 3.3.2, each guest of Xen shares a page for xenstore request and response ring buffers with Xenstore userspace application. It also binds an event channel with Xenstore application to send event notifications. Xenstore userspace application talks with Xen guests through ***Xen filesystem xenfs*** driver which mounts files for communication between Xen guests and userspace applications. Out of these files, following two are used for communication between Xenstore and Xen Dom0 guest:

- xsd_kva for mapping guest's page used for xenstore interface in userspace
- xsd_port for getting an unbound port number of the guest event channel used for binding it with a event port in userspace application i.e.Xenstore daemon.

For unprivileged guests of Xen, Dom0 controls their creation and performs management through Domain control hypercalls. While creation of DomU,a xenstore interface page and an unbound event channel are allocated which are introduced into xenstore daemon through Xen XL tool. DomU gets that allocated page for mappingit into its address space and the xenstore event channel with the help of hypercalls during its initialization.

In our current work, since no XL tool had been ported and Dom0 did not control creation of DomU guests, DomU xenstore page and event channel number was added manually. A similar file interface had been exposed to userspace by Dom0 which was used to map DomU xenstore page into Xenstore daemon application as shown in listing 5.4. Two globally shared static pages were allocated for xenstore interfaces of guests in PHIDIAS as shown in Figure 5.6. For event channel of DomU used with xenstore daemon, hard-coded DomU's event port number 1 was used. Mechanism for introducing DomU to Xenstore daemon with hypercalls and XL tool was replaced with by manually introducing creation of DomU connection into xenstore daemon code.

```
...
static int  xsd_foreign_kva_mmap (struct file *file , struct vm_area_struct
    *vma)
{
    size_t size = vma->vm_end - vma->vm_start;

    if ((size > PAGE_SIZE) || (vma->vm_pgoff != 0))
        return -EINVAL;

    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

    if (io_remap_pfn_range(vma, vma->vm_start,
                           0xfee45000 >> XEN_PAGE_SHIFT,
                           size, vma->vm_page_prot))
        return -EAGAIN;
```

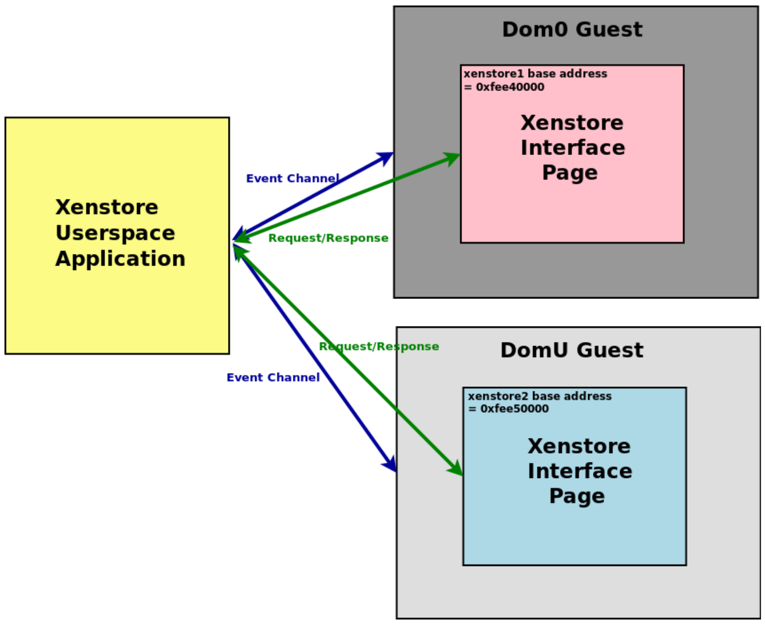


Figure 5.6: Communication between Xenstore application and Guests in PHIDIAS

```
    return 0;
}

const struct file_operations xsd_kva_foreign_file_ops = {
    .open = xsd_foreign_kva_open,
    .mmap = xsd_foreign_kva_mmap,
    .read = xsd_read,
    .release = xsd_release,
};

...
```

Listing 5.4: Added file interface for mapping DomU xenstore interface page into userspace in drivers/xen/xenfs/xenstored.c file

6 Porting Xen Network Virtualization

In this chapter, first Xen network virtualization architecture will be explained and then the work related to porting it to PHIDIAS will be presented. Other split drivers for different I/O devices can ported using the same approach adopted for paravirtualized network driver with some little modifications corresponding to the requirements for these split drivers.

6.1 Xen Network Virtualization

In this section, analysis of xen network virtualization architecture and data flow will be provided. The details of bringing up network virtual interfaces in domains will be explained. Most of the steps performed during registration and initialization of xen network drivers also applies to other I/O virtual drivers of xen. Figure 6.1 shows basic components of xen network virtualization architecture which are explained in next sections.

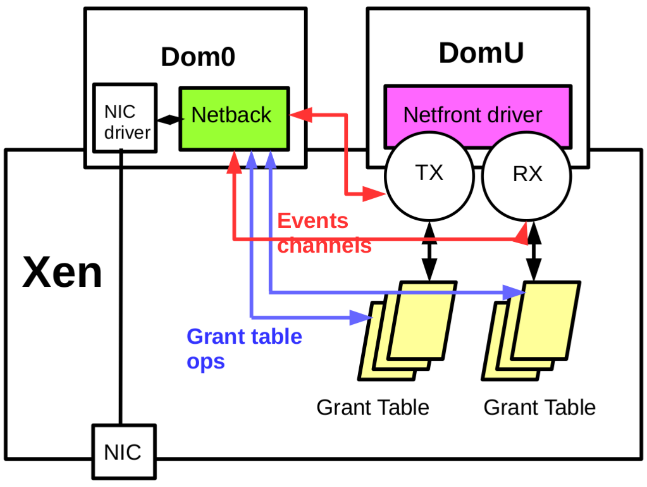


Figure 6.1: Xen network virtualization architecture

6.1.1 XenBus Backend and Frontend drivers

All split I/O drivers of Xen depends upon a general bus entity named XenBus which provides an interface to backend and frontend drivers to communicate with each other.

The internal working of Xenbus is dependent upon Xenstore and event channels. Xenbus is also composed to two split drivers that work together to provide successful connection of I/O split drivers as shown in Figure 6.2 .These are:

- **XenBus Backend** is a bus that itself is registered with kernel bus subsystem. It is responsible for enumerating all backend devices in Xenstore, calling their corresponding probe functions and watching xenstore for changes. All backend drivers register themselves with XenBus backend.

- **XenBus Frontend** is also a type of bus which registers itself with kernel bus subsystem. It is responsible for enumerating and probing all frontend devices in Xenstore and registering watches in Xenstore for getting notification about changes in the backend devices. All frontend drivers registert themselves with XenBus frontend.

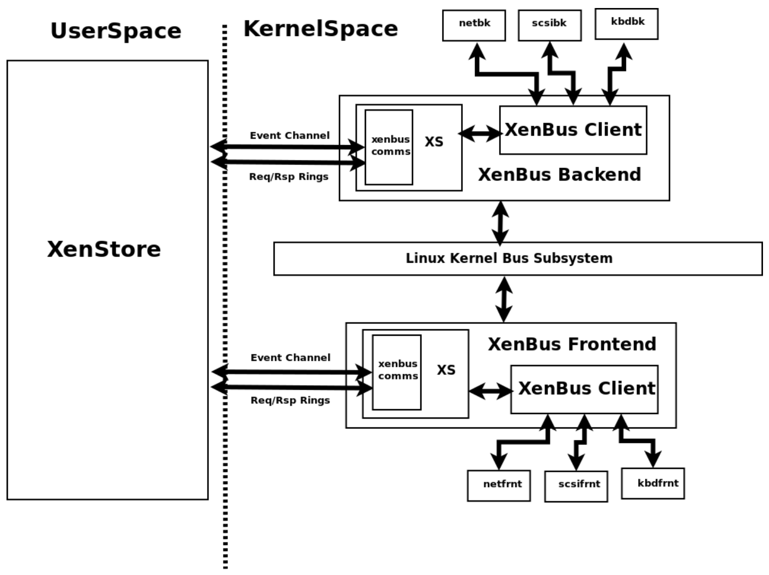


Figure 6.2: XenBus Architecture for connecting backends with frontends in Xen

6.1.2 Netback Driver

In Xen, Netback driver implements backend of split driver model in Dom0. It communicates with other end of split driver model through two rings i.e. transmit Tx ring and receive Rx ring. These rings are shared by network frontend driver and netback maps them into Dom0 address space. Netback driver is responsible for sending and receiving network packets to actual hardware through actual network driver of physical hardware.

6.1.3 Netfront Driver

Netfront driver implements the frontend of split driver model in unprivileged domains. It creates Tx and Rx rings for sending instructions about data flow to the backend. These rings do not create actual data packets. The data packets are transferred using shared memory pages offered by grant table mechanism of Xen. Both frontend and backend notifies each other of requests and responses through event channel.

6.1.4 Initialization of Network split drivers in Xen

Netback driver is probed when Xend toolstack in Dom0 creates **'vif'** device. In this netback_probe function, network backend sets its state to XenbusStateInitialising. It then write keys to the Xenstore related to features it supports for data processing. It includes feature-sg, feature-gso-tcpv4, feature-gso-tcpv6,feature-ipv6-csum-offload, feature-rx-copy,feature-rx-flip, feature-multicast-control, feature-dynamic-multicast-control,feature-split-event-channels, multi-queue-max-queues and feature-ctrl-ring. It then reads the main script provided in DomU XL configuration file for interfacing virtual network interface with real network device and kickoffs the necessary scripts to setup the desired connection between virtual interface and real network device. There are three main styles of network setup for a Xen host, bridged, routed and nat [44]. The default mode is bridged and from Xen 4.3 onwards, support of openvswitch is also provided.

Netfront driver gets probed by xenbus_probe function during its initialization only if Xenstore is up and running.Netfront probe calls xennet_create_dev to create a net_device and registers this device with Linux Network stack. It also allocates the transmit queue and receive queues for this net_device. After DomU is started and netfront driver is initialized successfully, hotplug scripts in Dom0 sets the initial state of netfront to XenbusStateInitialising by writing corresponding netfront state key in Xenstore. Both netback and netfront has registered a notifier with Xenstore through Xenbus which watches the state of other end and get notifications from Xenstore upon changes in state of other end. Setting of the state of netfront to XenbusStateInitialising causes netback to get notified and it changes its Bus state to XenbusStateInitWait. When netfront driver gets notified about XenbusStateInitWait of netback, it calls xennet_connect function. xennet_connect is the main function which does all the necessary work to connect netfront with netback. It creates tx and rx shared rings, grants the access right to netback, allocates tx and rx event channels and store tx-ring-ref, rx-ring-ref, event-channel-tx, event-channel-rx request-rx-copy, feature-rx-notify, feature-sg and feature-gso-tcpv4 to xenstore. It also allocates rx buffers for receiving packets from netback. After completing its talk with netback, it sets its state to XenbusStateConnected.

On receiving state change notification about successful connection from netfront, netback reads the keys written by netfront. It reads grant refs of tx and rx rings and maps them into Dom0 address space. It binds its event channels to netfront tx and rx event channels. It then finally sets its state to XenbusStateConnected.

6.1.5 Network Data Flow from Netfront to Netback

For transmitting network packets to netback, netfront calls xennet_start_xmit function. It examines received skb_buff structure which is used by kernel to manage network packets. First, it allocates grant reference and tx request for linear part of skb_buff. Then it calculates the number of fragments in network packet and allocates grant reference and tx request for all fragments of skb_buff. Finally, it notifies netback driver of tx requests in Tx ring.

On getting notification from netfront about tx requests, netback processes the requests in xenvif_tx_action and creates corresponding copy and map operations to be performed on shared grant references. Netback driver has the maximum length of TX copy operation to be 128 bytes. If the packet size is larger than 128 bytes, remaining data will be mapped into Dom0 address space. After filling a newly allocated skb_buff with received network packet data, netback gives this packet to Linux network stack which forwards it to upper layers.

6.1.6 Network Data Flow from Netback to Netfront

For transmission of packets from netback to netfront, xenvif_start_xmit is called. It first checks the number of available of rx buffer slots in RX ring of netfront mapped into Dom0 address space. After getting required number of slots in RX ring, it copies the packet data into rx buffers granted by netfront driver through gnttab_batch_copy function. This in turn calls hypercall responsible for transferring data into netfront rx buffers. Netback driver uses this batch copy operation to copy data to and from an unprivileged guest via the grant references in the RX and TX ring buffers. Netfront driver calls xennet_poll function to retrieve the network data and forwards it to Linux network stack.

6.2 Network Virtualization Architecture in PHIDIAS

In this section, changes required for porting Xen network split drivers to PHIDIAS will be discussed.

6.2.1 Writing Keys to Xenstore for network virtualization

In Xen, Xenstore should be running for split drivers of I/O devices to initialize and connect with each other. In case of PHIDIAS, Xenstore was cross-compiled using build root method as described in Xen wiki page [45]. It was started in Dom0 Linux guest using the commands in listing 6.1.

```
#mount -t xenfs xenfs /proc/xen
#cp -r /usr/local/lib/* /lib
#mkdir /var/run
#touch /var/run/xenstored.pid
#touch /var/lib/xenstored/tdb
#chmod +x /usr/local/sbin/xenstored
```

```
#./usr/local/sbin/xenstored --pid-file /var/run/xenstored.pid --priv-domid
    1
```

Listing 6.1: Commands for startin Xenstore dameon in Dom0 on PHIDIAS

First command mounts the Xen filesystem so that userspace applications can get access to shared Xenstore pages of Dom0 and DomU guests.It also creates a file for accessing event channel number of Dom0 to bind with Xenstore's event channel. The last command basically starts the Xenstore in daemon mode and gives DomU with ID 1 access rights to write/read keys in its Xenstore page. By default, only Dom0 has permissions to write or modify keys in Xenstore. It can grant permission to other guests by using **xenstore-chmod** command but it would require for Dom0 to know about all the keys which DomU would need to read/write/modify in Xenstore. The simpler approach is to start Xenstore daemon with **priv-domid 1** which will grant guest with Domain ID 1 permissions to read/write/modify all keys in Xenstore .

6.2.2 Probing Netback driver

As described in section 6.1.4, netback driver is probed by Xend toolstack . In case of our setup, since there was no Xend toolstack, it gets probed by Xenstore dameon by notifying the Dom0 guest after being started successfully. No modification was done in the code as Xen already supports probing netback by Xenstore. However, there are some keys which were written by Xend and Xendomains services in Dom0 on native Xen setup. These were manually written to Xenstore by Dom0 guest by using the script in listing 6.2.

```
#!/bin/bash

cd /usr/local/bin/
chmod +x *

./xenstore-write /local/domain/0/domid 0
./xenstore-write /local/domain/0/name Domain-0
./xenstore-write /local/domain/0/control/shutdown ""
./xenstore-write /local/domain/0/control/feature-poweroff 0
./xenstore-write /local/domain/0/control/feature-halt 0
./xenstore-write /local/domain/0/control/feature-suspend 0
./xenstore-write /local/domain/0/control/feature-reboot 0
./xenstore-write /local/domain/1/device ""
./xenstore-write /local/domain/1/device/vif ""
./xenstore-write /local/domain/1/device/vif/0 ""
./xenstore-write /local/domain/1/device/vif/0/backend-id 0
./xenstore-write /local/domain/1/device/vif/0/mac "D2:A3:CD:27:4A:53"
./xenstore-write /local/domain/1/device/vif/0/backend "/local/domain/0/
    backend/vif/1/0"
./xenstore-write /local/domain/0/backend/vif/1/0/frontend-id 1
./xenstore-write /local/domain/0/backend/vif/1/0/frontend "/local/domain/1/
    device/vif/0"
./xenstore-write /local/domain/0/backend/vif/1/0/script "/etc/xen/scripts/
    vif-route"
```

```
./xenstore-write /local/domain/0/backend/vif/1/0/handle 0
./xenstore-write /local/domain/0/backend/vif/1/0/mac "D2:A3:CD:27:4A:53"
./xenstore-write /local/domain/1/device/vif/0/state 1
./xenstore-write /local/domain/0/backend/vif/1/0/state 1
```

Listing 6.2: Script used in Dom0 to write necessary keys for network drivers to Xenstore on PHIDIAS

6.2.3 Probing Netfront driver

As described in section 6.1.4, netfront driver gets probed by XL toolstack run in Dom0 while creating DomU. In PHIDIAS, both Dom0 and DomU guests start at the same time. Netfront driver should only probed after Xenstore and netback driver are running successfully. To notify DomU guest about running Xenstore daemon in Dom0 and probing netback driver, an interprocess interrupt was used. An ipc capability with index 2 was added in configuration options of Dom0 linux guest in PHIDIAS as shown in listing 6.3.

```
    <cap type="ipc" target_xref="linux2" param="0xa" />
```

Listing 6.3: Capability added in config options of Dom0 linux guest to notify DomU guest to probe netfront driver

SGI number 10 was used. In netfront driver, an ipi handler, listed in 6.4, was registered which was called on receiving interrupt from Dom0.

```
static irqreturn_t irqHandlerBusProbe (int irq, void *dev_id)
{
   schedule_work(&probe_work);
   return IRQ_HANDLED;
}
```

Listing 6.4: IPI handler for receiving notification from Dom0 about successful running of Xenstore and netback driver

To send an IPI to DomU from Dom0, a small kernel module was developed which triggered the capability with index 2. This module was inserted after running Xenstore daemon and writing network related xenstore keys in Dom0 guest. This module triggers the capability by using the code described in listing 6.5.

```
        unsigned int val = 2;
        asm volatile("mov x0, #0x9999\n\tmov x1, %0\n\thvc #0" :: "r" (val)
            : "x0", "x1");
```

Listing 6.5: Code to trigger capability 2 from Dom0 to DomU

6.2.4 Transmission of packets from netfront to netback

For transmission path from netfront to netback, grant entries are created in Tx ring by netfront which contain page frame numbers of shared pages containing network packet data. As described in section 5.2.2, a ZONE_XEN was created for sharing memory pages between frontend and backend drivers. Since skb_buff was allocated from linux kernel **NORMAL** memory zone in native Xen guest's netfront driver, it had to copied into ZONE_XEN so that it could be shared with Domain0. For this, a function xen_skb_copy to copy skb_buff from NORMAL memory zone to XEN zone was implemented in skbuff.c file and called in xennet_start_xmit which is main function for transmitting packets to Dom0. Linux kernel network stack provides a function skb_copy for copying both an sk_buff head and its data from non-linear to linear buffers. This had been used as a reference for implementation of xen_skb_copy which is provided in listing 6.6 along with the way it is called from xennet_start_xmit function .

```
File : net/core/skbuff.c

struct sk_buff *xen_skb_copy(const struct sk_buff *skb, gfp_t gfp_mask)
{
    int headerlen = skb_headroom(skb);
    unsigned int size = skb_end_offset(skb) + skb->data_len;
    struct sk_buff *n = __xen_alloc_skb(size, gfp_mask);
    if (!n)
        return NULL;

   /* Set the data pointer */
    skb_reserve(n, headerlen);
   /* Set the tail pointer and length */
    skb_put(n, skb->len);

    if (skb_copy_bits( skb, -headerlen, n->head, headerlen + skb->len))
        BUG();

    copy_skb_header(n, skb);
    return n;
}
 ..
 File : drivers/net/xen-netfront.c

 static int xennet_start_xmit(struct sk_buff *skb, struct net_device *dev)
 {
 ...
     struct sk_buff *nskb;
     struct sk_buff *xen_skb;

   /* Drop the packet if no queues are set up */
    if (num_queues < 1)
        goto drop;

    xen_skb = xen_skb_copy(skb, GFP_XEN);
    if (!xen_skb)
        goto drop;

    dev_kfree_skb_any(skb);
    skb = xen_skb;
    queue_index = skb_get_queue_mapping(skb);
    queue = &np->queues[queue_index];
...
}
```

Listing 6.6: Code snippet of function for copying entire network packet from NORMAL memory to XEN memory zone

This copying from NORMAL zone to XEN zone for sharing packets would definitely degrade the networking performance for larger size packets which would be visible while performing throughput testing in later chapter 7.

6.2.5 Receiving packets from netfront by netback

In native Xen setup, Netback driver uses GNTTABOP_map_grant_ref hypercall for mapping shared network packets into its domain. This hypercall operation adds a mapping at host virtual address in the current address space of Dom0. Dom0 allocates pages in its address space and provides virtual addresses of those pages as host address in GNTTABOP_map_grant_ref hypercall. Xen hypervisor then creates a mapping of shared pages at those host virtual addresses in Dom0 page tables. Since page table entries are added by hypervisor for accessing shared network packets by Dom0, netback driver uses virt_to_page macro on these mapped pages while filling skb_buff fragments of received packets. Linux network stack uses struct page's in most of APIs for handling fragments of network packets. The macro virt_to_page returns struct page's of given virtual address which is called in netback function xenvif_fill_frags to fill skb_buff fragments by _skb_fill_page_desc function.

In our ported setup, shared pages are mapped using ioremap which maps provided physical addresses into caller's address space. The macro virt_to_page cannot be used for virtual addresses returned by ioremap. Dom0 only allocates struct page's for the memory given to it by PHIDIAS. It has no struct page's for DomU's shared memory, otherwise its kernel's buddy allocator [46] can allocate pages from DomU's memory. In order to solve this problem, netback driver in PHIDIAS maps the shared network data pages into its address spaces and then copies data into its newly allocated pages of NORMAL memory zone. It then forwards these to Linux network stack for further processing. This has also degraded the network performance on PHIDIAS as shown in later chapter 7.

6.3 Porting other virtual I/O devices in Xen

In this section, steps require in porting of Xen I/O virtual devices ,besides network, to PHIDIAS will be discussed. One of the core virtual devices in Xen is block device which provides a non-volatile storage to guests to store and retain data between power reboots.

6.3.1 Xen Virtual Block Device Driver

Xen virtual block device driver provides an interface to an abstract block device, typically a virtual hard disk backed by real disks, individual partitions, or even files on a host filesystem [8]. Just like other split drivers, it consists of block frontend and backend drivers and it is based on grant table sharing and event channels mechanisms of Xen hypervisor. Grant table operations are used for transferring several KB of data consisting of multiple blocks. It provides an abstraction of a block device similar to SATA or SCSI with general read and write block device operations. It also supports command-reordering which means that commands might complete in an order different from the order in which they are issued.

As grant table and event channels are already ported to PHIDIAS, the only thing which needs to be done for setting up Xen's block virtual device is writing keys to Xenstore which are read by block frontend driver while finalizing the connection with backend. The frontend driver reads domain's device/vbd/0/backend key in the XenStore to get the location of back end for the virtual block device. The most important information it needs to read from Xenstore is related to sector size and number of sectors which is provided in listing 6.7.

```
/local/domain/0/backend/vbd/1/0/frontend-id 1
/local/domain/0/backend/vbd/1/0/frontend "/local/domain/1/device/vbd/0"
/local/domain/0/backend/vbd/1/0/sector-size
/local/domain/0/backend/vbd/1/0/size
/local/domain/0/backend/vbd/1/0/info

/local/domain/1/device/vbd/0/backend-id 0
/local/domain/1/device/vbd/0/backend "/local/domain/0/backend/vbd/1/0"

/local/domain/1/device/vbd/0/state 1
/local/domain/0/backend/vbd/1/0/state 1
```

Listing 6.7: Keys in Xenstore for connecting virtual block device frontend driver with backend in Xen

As with other split drivers, block front end driver allocates a page for ring buffer for granting access to block backend driver. It writes ringref of shared ring page to Xenstore. It also allocates and event channel and passes it to backend through Xenstore. It reads Xenbus state of backend and sets it own state while connecting itself with backend. States are changed at different steps of talking with the other end until XenbusStateConnected is reached and set.

In PHIDIAS setup, since both guests are started at the same time unlike Dom0 and DomU in Xen, block frontend driver should be probed through an IPI triggered by block backend after successful running of Xenstore daemon and block backend probing in Dom0 guest. The same approach used for netfront driver, as described in section 6.2.3, could be followed.

6.3.2 Porting miscellaneous Xen I/O Device Drivers

In this thesis, only network virtualization drivers of Xen had been ported to PHIDIAS. For remaining I/O devices e.g, keyboard, mouse, scsi, block devices etc, the necessary Xenstore keys read by frontend drivers should be written manually to Xenstore filesystem through a customized script from Dom0 guest. User could locate the necessary keys needed by frontend drivers by looking through their driver code. For probing the frontend drivers, an IPI can be configured and triggered by a kernel module inserted into Dom0 Linux guest.

7 Testing and Evaluation

In this chapter, results of tests performed for comparing network performance on Xen and PHIDIAS will be presented and analyzed. Two tests were performed for calculating statistics on network performance on Xen and PHIDIAS. One test was conducted to calculate the latency of transferring network packets between Dom0 and DomU and the second one was done to measure throughput. Both tests were performed by running two Linux guests on different physical CPU cores on an 8-core Hikey ARMv8 target platform. Analysis and results of these tests are presented in following sections.

7.1 Network Latency Test

Ping is the most common utility to measure round trip latency of network packets. For our tests, Ping binary used was of **BusyBox v1.22.1 (Debian 1:1.22.0-9+deb8u1) multi-call binary**. Ping utility was obtained from prebuilt binary of ramdisk image of 96boards' linaro debian at web link [47].

Ping tests were performed with different packet sizes using default Ethernet maximum transmission unit (MTU) i.e. 1500 bytes. Each test was conducted for 50 packets and then minimum, average and maximum round trip latencies were calculated. For TCP/IP networking, if network packet size is larger than MTU, IP fragmentation occurs [48]. For testing fragmentation in our setup, packet size of 1900 bytes was used.

7.1.1 Network Latency Test results on Xen

Table 7.1 shows the results of ping tests performed between Dom0 and DomU through virtual network devices on Xen.

Table 7.1: Ping Test results on Xen

Number of packets	Size of data in Bytes	Min RTT (ms)	Avg RTT (ms)	Max RTT (ms)	Packets Lost	Reason
50	56 (default)	0.42	0.533	0.718	0	N/A
50	1000	0.402	0.611	0.857	0	N/A
50	1900 (with fragmentation)	0.483	0.699	0.913	0	N/A

7.1.2 Network Latency Test results on PHIDIAS

Table 7.2 shows the results of ping tests performed between Dom0 and DomU through virtual network devices on PHIDIAS.

Table 7.2: Ping Test Results on PHIDIAS

Number of packets	Size of data in Bytes	Min RTT (ms)	Avg RTT (ms)	Max RTT (ms)	Packets Lost	Reason
50	56 (default)	0.135	0.147	0.36	0	N/A
50	1000	0.234	0.251	0.558	0	N/A
50	1900 (with fragmentation)	0.394	0.407	0.427	30	IP reassembly Timeout

7.1.3 Analysis of Network Latency Test results on PHIDIAS

7.2 Network Throughput Test

7.2.1 Network Throughput Test results on Xen

7.2.2 Network Throughput Test results on PHIDIAS

7.2.3 Analysis of Network Throughput Test results on PHIDIAS

8 Conclusion

Bibliography

[1] http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dui0379c%2FCHDEDCCD.html.

[2] http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.den0024a%2FCDDEIJCH.html.

[3] http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.den0024a%2Fch12s04.html.

[4] J. Nordholz, "Design and provability of a statically configurable hypervisor," Jun 2017. https://depositonce.tu-berlin.de/handle/11303/6388.

[5] "Xen," Oct 2017. https://en.wikipedia.org/wiki/Xen.

[6] "Xen arm with virtualization extensions whitepaper." https://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper.

[7] T. L. F. Follow, "Linaro connect : Introduction to xen on arm," Mar 2013. https://www.slideshare.net/xen_com_mgr/linaro-connect-xen-on-arm-update?qid=d094f87d-20d7-4577-a884-7d749bf666d1&v=&b=&from_search=10.

[8] D. Chisnall, *Definitive guide to the xen hypervisor*. Prentice Hall, 2013.

[9] T. L. F. Seguir, "Xpds13: "unlimited" event channels - david vrabel, citrix." https://pt.slideshare.net/xen_com_mgr/unlimited-eventchannels.

[10] Y. C. Cho and J. W. Jeon, "Sharing data between processes running on different domains in para-virtualized xen," in *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*, pp. 1255–1260, IEEE, 2007.

[11] "an introduction to virtualization." ttp://www.kernelthread.com/publications/virtualization/.

[12] "Reasons to use virtualization." https://docs.oracle.com/cd/E26996_01/E18549/html/BHCJAIHJ.html.

[13] "Understanding full virtualization, paravirtualization, and hardware assist," Jun 2017. https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html.

[14] H. Lee, "Virtualization basics: Understanding techniques and fundamentals,"

[15] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Proceedings of the fourth symposium on Operating system principles - SOSP 73*, 1973.

[16] "What's the difference between type 1 and type 2 hypervisors?."

[17] "Top 10 virtualization technology companies for 2016." `http://www.serverwatch.com/server-trends/slideshows/top-10-virtualization-technology-companies-for-2016.html`.

[18] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 9–16, IEEE, 2011.

[19] P. Koopman, "Design constraints on embedded real time control systems," 1990.

[20] S. Heath, *Embedded systems design.* Newnes, 2005.

[21] A. Aguiar and F. Hessel, "Embedded systems virtualization: The next challenge?," *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, 2010.

[22] "Virtualization for embedded systems," Apr 2011.

[23] "Arm architecture," Sep 2017. `https://en.wikipedia.org/wiki/ARM_architecture`.

[24] ARM, "Architecture arm developer." `https://developer.arm.com/products/architecture`.

[25] "arm architecture reference manual armv8 for armv8-a architecture profile beta," Sep 2013. `https://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf`.

[26] "Arm cortex-a53 mpcore processor revision: r0p4," Feb 2013. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/DDI0500G_cortex_a53_trm.pdf`.

[27] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you." `https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`.

[28] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos, "Arm virtualization," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, p. 304316, 2016. `http://www.cs.columbia.edu/~cdall/pubs/isca2016-dall.pdf`.

[29] "Opensynergy." `http://www.opensynergy.com/produkte/coqos/`.

[30] "Xen project software overview." `https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview`.

[31] S. J. Vaughan-Nichols, "Xen becomes a linux foundation project," Dec 2015. http://www.zdnet.com/article/xen-becomes-a-linux-foundation-project/.

[32] X. Wang, Y. Sun, Y. Luo, Z. Wang, Y. Li, B. Zhang, H. Chen, and X. Li, "Dynamic memory paravirtualization transparent to guest os," *Science China Information Sciences*, vol. 53, no. 1, pp. 77–88, 2010.

[33] "Xen project release features," Aug 2017. https://wiki.xenproject.org/wiki/Xen_Project_Release_Features.

[34] "Event channel internals," Sept 2014. https://wiki.xen.org/wiki/Event_Channel_Internals.

[35] "Xl," Nov 2016. https://wiki.xenproject.org/wiki/XL.

[36] "Xenstore," Feb 2015. https://wiki.xen.org/wiki/XenStore.

[37] "Xenstore reference," Feb 2015. https://wiki.xen.org/wiki/XenStore_Reference.

[38] "Xenbus," june 2014. https://wiki.xen.org/wiki/XenBus.

[39] "Hikey (lemaker version) general — specification — quick start — wiki — faq — resources." http://www.lemaker.org/product-hikey-specification.html.

[40] 96boards, "96boards/linux," Jan 2016. https://github.com/96boards/linux/tree/android-hikey-linaro-4.1.

[41] https://www.kernel.org/pub/linux/kernel/v4.x/testing/.

[42] http://xenbits.xen.org/gitweb/?p=xen.git;a=commit;h=2831f2099b6175384817d7afd952f7918998b39a.

[43] Michal Simek, "The official linux kernel from xilinx - merge tag 'v4.9' into master." https://github.com/Xilinx/linux-xlnx/blob/master/arch/arm/kernel/smp.c.

[44] "Network configuration examples (xen 4.1+)," Aug 2016. https://wiki.xenproject.org/wiki/Network_Configuration_Examples_(Xen_4.1%2B).

[45] "Xen arm with virtualization extensions/crosscompiling," April 2017. https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions/CrossCompiling.

[46] "Chapter 6 physical page allocation." https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[47] S. Sargunam. https://builds.96boards.org/releases/hikey/linaro/debian/latest/initrd.img-3.18.0-linaro-hikey.

[48] “Ip fragmentation,” June 2017. `https://en.wikipedia.org/wiki/IP_fragmentation`.