Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Security in Telecommunications

Masterarbeit

Porting the Xen Emulation Layer to static hypervisor

presented by
Amna Waseem
student ID: 387424
student of
Masterstudiengang ICT Innovation
Embedded Systems

for obtaining the academic degree
Master of Engineering

Betreuender Hochschullehrer: Prof. Dr. Jean-Pierre Seifert
Betreuender Mitarbeiter: Dipl.-Inf. Robert Buhren

Berlin
November 11, 2017

Acknowledgments

I would like to thank Robert Buhren for giving me this opportunity to work on a very exciting topic for my Master Thesis and introducing me to a new challenging domain of virtualization. The door to his office was always open whenever I ran into a problem or had any questions related to research. He had allowed me to work independently on designing implementation strategies of research, but guided me in the right direction whenever he thought I needed it. I also want to thank everybody at Security in Telecommunications group at TU Berlin for the wonderful time, especially, Jan Nordholz for sharing his wisdom and answering all my questions.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbststndig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, November 11, 2017

Amna Waseem

Abstract

Over the past decade, virtualization technologies have gone from server consolidation in corporate data centers to small forms of embedded platforms for providing 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 usage of physical resources. Major hypervisor vendors in virtualization market have been continuously developing different 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 (Provable Hypervisor with Integrated Deployment Information and Allocated Structures) is a statically configured 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 reduced memory footprint by eliminating dynamic elements from the system. However, it does not have support of I/O device virtualization. In this thesis, Xen Hypervisor's I/O device framework has been ported to PHIDIAS using its existing mechanisms of static memory sharing and inter-guest communication. Working and testing of ported virtualized network devices have been shown. Results are gathered for comparing throughput and bandwidth of virtual network interfaces on native Xen and PHIDIAS setup. Results highlight the fact that PHIDIAS delivers better performance for packets without fragmentation by bypassing the hypervisor layer for handling Xen specific hypercalls. However,current approach in ported network drivers introduces overhead while manually copying fragmented packets since PHIDIAS has no hypercall implemented for mapping shared pages to guest domain space by modifying corresponding page table entries as compared to Xen.

Keywords. I/O virtualization, Hypervisor, Xen, Static configuration, ARM architecture

Zusammenfassung

In den letzten zehn Jahren sind Virtualisierungstechnologien von der Serverkonsolidierung in Unternehmens-Datenzentren hin zu kleinen Formen von eingebetteten Plattformen fr Systemsicherheit, Hardwareisolierung, Ressourcenverwaltung und E / A-Gerteemulation gegangen. Mit den modernen Computern, die verschiedene Formen von I / O-Gerten bereitstellen, hat es eine enorme Entwicklung in der Hypervisor-Technologie gegeben, um eine Effizienz bei der Verwendung von physischen Ressourcen bereitzustellen. Groe Hypervisor-Anbieter im Virtualisierungsmarkt haben kontinuierlich verschiedene Techniken entwickelt, um Speicher dynamisch zuzuweisen und Ressourcen effizienter zu nutzen. Diese Dynamizitt bei der Speicherzuweisung fhrt jedoch zu Schwier-igkeiten bei der Verifizierung und Verifizierung des konfigurierten Hypervisors, der in sicherheitskritischen Anwendungen verwendet wird. PHIDIAS (Proviable Hypervisor mit Integrated Deployment Information und Allocated Structures) ist ein statisch konfigurierter Hypervisor, der von der Abteilung Sicherheit in der Telekommunikation der Fakultt fr Elektrotechnik und Informatik der Technischen Universitt Berlin entwickelt wurde und das Prinzip der Statik zur Erleichterung der Beweisbarkeit und zur Reduzierung der Laufzeitkomplexitt beinhaltet. mit reduziertem Speicherbedarf durch Eliminierung dynamischer Elemente aus dem System. Die E / A-Gertevirtualisierung wird jedoch nicht untersttzt. In dieser Arbeit wurde das I / O-Device-Framework von Xen Hypervisor unter Verwendung der vorhandenen Mechanismen der statischen Speicherfreigabe und der Inter-Guest-Kommunikation nach PHIDIAS portiert. Das Arbeiten und Testen von portierten virtualisierten Netzwerkgerten wurde gezeigt. Die Ergebnisse werden gesammelt, um den Durchsatz und die Bandbreite virtueller Netzwerkschnittstellen bei nativem Xen- und PHIDIAS-Setup zu vergleichen. Die Ergebnisse unterstreichen die Tatsache, dass PHIDIAS eine bessere Leistung fr Pakete ohne Fragmentierung liefert, indem die Hypervisor-Schicht zur Behandlung von Xen-spezifischen Hypercalls umgangen wird. Der aktuelle Ansatz bei portierten Netzwerktreibern fhrt jedoch zu einem Overhead, whrend fragmentierte Pakete manuell kopiert werden, da PHIDIAS kein Hypercall implementiert hat, um gemeinsam genutzte Seiten in den Gastdomnenbereich zu mappen, indem entsprechende Seitentabelleneintrge im Vergleich zu Xen modifiziert werden.

Keywords. I/O-Virtualisierung, Hypervisor, Xen, statische Konfiguration, ARM-Architektur

Contents

List of Figures

List of Tables

1. Introduction

With a diverse range of applications and market opportunities of virtualization technology, it has become one of leading technologies of the future. Though virtualization began its journey from mainframes, it soon found its way into servers and desktop virtualization. By keeping its continuous improvements, it has now entered into today's most demanding market space of embedded systems.

A few decades ago, embedded systems were designed to run simpler applications with specific design constraints. But the present embedded systems are able to provide complex functionality, targeting real time applications and defense mission critical systems and support a diverse range of I/O devices. Providing optimal, flexible and robust hardware utilization of these complex I/O devices was a challenge faced by virtualization industry in recent years. For this purpose, efforts of virtualization industry has been focused on developing an efficient software, called hypervisor, that could create several different virtual machines hosting on a single platform and provide memory management, resource allocation and sharing of hardware devices in addition to the secure execution of those guest machines running on top of it. In short, we can say that running multiple mixed-criticality applications on embedded systems, providing intelligent I/O virtualization and ensuring their security were the major sources of evolution in virtualization technology.

In the field of developing efficient software for better hardware utilization, multiple players are providing innovative solutions to meet the needs of virtualization industry. According to 2016 Spiceworks State of IT report [11], more than 76% of organizations are taking advantage of virtualization today, and that number will continue to grow in the future. Xen and KVM are amongst the most popular open source hypervisors available in market. Both provide support for I/O device virtualization. However, they both reply upon dynamic memory and resource allocation techniques which makes it harder to prove their functional correctness which is a crucial requirement for the reliability of safety- and security-critical systems.

PHIDIAS is a hypervisor developed by Dr.-Ing. Jan Nordholz at TU Berlin Telekom Innovation Laboratories [4],supporting ARM, MIPS and x86. It is based on static compile time configuration of virtual machines to remove non-mandatory dynamic components from the system. This static configuration feature of PHIDIAS reduces dynamicity from the system which results in a simpler provable code with a smaller memory footprint. However, it only provides virtualization of timer and UART devices. There is no support for other virtual I/O devices e.g. network and block devices.

Theoretically speaking, adding support of I/O virtualization in PHIDIAS could be done in two ways. One way is to write virtual device drivers for I/O devices from the scratch. And another method is to use I/O virtualization model of an already existing hypervisor and use it as a reference for porting those virtual device drivers to PHIDIAS.For our work, second method seems to be a reasonable approach as it would require less time. Now between Xen and KVM as two promising choices for using as a reference, Xen seems to be a better option. Most important reasons are that it is a thin type-1 or bare metal hypervisor with less areas of attacks and makes guests OSes responsible for implementing and maintaining virtual I/O drivers. On the other hand, KVM is a hosted or type-2 hypervisor which requires target processor to support hardware virtualization and uses VirtIO model [1] where backend drivers of PV I/O devices reside inside hypervisor [**?**]. A detailed view of Xen will be explained in this thesis for the better understanding of its I/O architecture.

This thesis will add support of I/O virtualization in PHIDIAS by using Xen I/O architecture as a reference. Due to time restrictions, it will focus on running and testing virtual network devices on PHIDIAS and compare network performance in terms of throughput and bandwidth. Methods for running other I/O devices will be discussed theoretically along with benefits and areas of future research.

1.1. Motivation

There are two main motivational factors behind this work. First is to demonstrate the fact that even though PHIDIAS is a statically configured hypervisor with a minimal set of drivers, its structure is flexible enough to add support for other virtual I/O devices. And the second motivation is to compare the performance of virtualized network I/O devices between Xen and PHIDIAS.

1.2. Aims and Objectives

Main aims of the current work are twofold. First is to investigate mechanisms in PHIDIAS for memory sharing and interprocessor communication and extending its feature set to accommodate ported virtual I/O devices. Second is to keep changes in Xen I/O drivers and PHIDIAS as minimum as possible in order to reuse maintenance support of Xen community and to keep the smaller footprint and static nature of PHIDIAS intact.

For achieving current aims, this thesis will focus on finding core building blocks of Xen I/O virtualization model and replace them with corresponding components of PHIDIAS.

[1] VirtIO is a virtualization standard for network and disk device drivers where just the guest's device driver "knows" it is running in a virtual environment, and cooperates with the hypervisor. This enables guests to get high performance network and disk operations, and gives most of the performance benefits of paravirtualization [12].

For running and testing the ported system, ARM will be used as a target platform due to its wide range of applications in smart phones, laptops, digital TVs, set-top boxes, mobile, and Internet of Things 'IoT' devices.

1.3. Related Work

There has been a lot of work on developing techniques for high-performance I/O virtualization. Over the past decade, achieving better hardware utilization by decoupling logical I/O devices from physical ones has been the focus of researchers in virtualization domain.
Gordon et al. [13] proposed an exitless paravirtual I/O model, under which guests and the hypervisor, running on distinct cores, exchange exitless notifications instead of costly exit-based notifications for increasing the performance of paravirtual I/O. Nadav et al. [14] developed ELVIS (Efficient and scaLable paraVirtual I/O System) for running host functionality on dedicated cores that were separate from guest cores, and thus avoiding exits on the I/O path. Shafer et al. [15] introduced concurrent direct network access (CDNA), a new I/O virtualization technique combining both software and hardware components that significantly reduces the overhead of network virtualization in VMMs. With CDNA, a unique context is allocated to each virtual machine on the network interface for communicating directly with the network interface through that context. In this way, tasks of traffic multiplexing, interrupt delivery, and memory protection gets divided between hardware (network interfaces) and software (virtual machine monitor). Shinagawa, Kawai and Kono et al. [16] developed a small hypervisor named BitVisor for the fast I/O access from the guest operating systems by passing-through the virtual machine monitor.

In addition to developing innovating software techniques for efficient virtualization of I/O subsystems and peripheral devices, researchers have also worked on adding hardware support for virtualization in I/O devices. The idea is to bypass the hypervisor by virtual machines to directly access logical interfaces implemented on actual hardware, thus producing best performance. Himanshu and Karsten [17] developed a new approach of I/O virtualization by offloading select virtualization functionality onto the device termed as self-virtualized devices. Intel's has developed Single Root I/O Virtualization (SR-IOV) technology [18] that extends the physical function of the I/O port of an I/O device to a number of virtual functions directly assigned to virtual machines, hence achieving near native performance without software emulation.

Broadly speaking, efforts for developing effective I/O virtualization techniques have resulted in three types of solutions i.e software-based, hardware-based and combination of both software and hardware based I/O virtualization. On one side, software-based approach, uses paravirtualized drivers implemented in dedicated driver domain or as a part of hypervisor and provides isolated execution of I/O virtual devices with live migration [19] and traffic monitoring [20] support for virtual machines. On the other side,

hardware method reduces the software overhead and bypass hypervisor layer to achieve good performance. In the middle, there is a combination of both approaches. Research shows that all solutions have their pros and cons and the selection heavily depends upon the target application.

1.4. Thesis structure

Following this section the thesis will be structured as follows:

Chapter 2 will present some background information on virtualization technologies, embedded systems with ARM architecture and a brief overview of PHIDIAS.

Chapter 3 will have an in depth look at the architecture of Xen hypervisor and the core components of its split device driver model for I/O virtualization.

Chapter 4 will shed light on basic setup of working environment and discuss requirements and assumptions of thesis.

Chapter 5 will feature a more thorough presentation of porting basic components of Xen split driver model while **Chapter 6** will describe porting of Xen network drivers to PHIDIAS.

Chapter 7 will be about testing and evaluating the network performance on Xen and PHIDIAS. The results of latency and bandwidth tests will be presented and analyzed.

Chapter 8 will summarize the thesis along with potential limitations and benefits, and gives an outlook on future work.

Appendix **??** features patch for the changes in source code of Linux guests and PHIDIAS related to current work in addition to some instructions for running virtual network devices and running network tests on PHIDIAS and Xen.

2. Background

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

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 and 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. This technology has dramatically changed the IT landscape by reducing underlying 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 [21].

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 vendors 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 a king of x86 server virtualization. However, many players e.g. Citrix and Microsoft started to gain ground in this field by offering additional middle-ware and desktop virtualization solutions. Over the past several years, there has been huge improvements in this domain and vendors are continuously innovating to increase capabilities of virtualized systems and developing their management tools.

Following are some of the main benefits of using virtualization technology [22]:

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 execution of other guests.

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 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 the 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 [23]. It is an abstraction layer that enables communication between hardware and virtual machines [24]. Figure 2.1 provides an illustration of virtualization architecture.

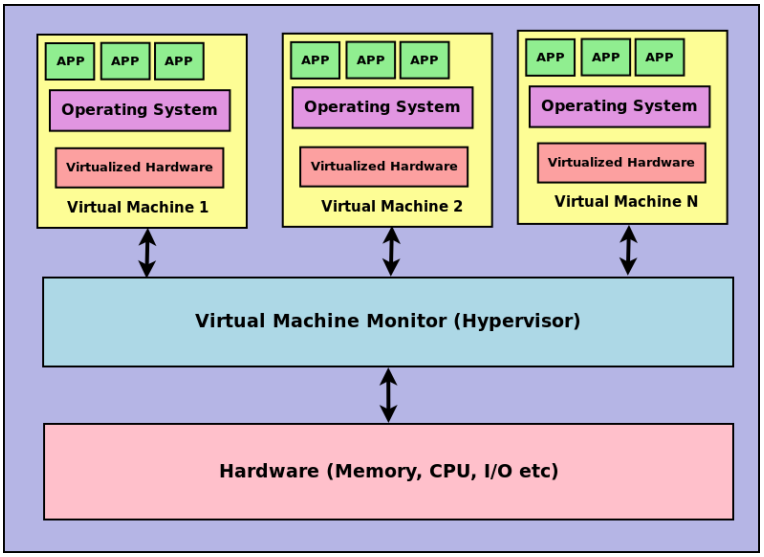

Figure 2.1.: Virtualization Framework

In 1974, Popek and Goldberg published an article 'Formal Requirements for Virtualizable Third Generation Architectures' in which they described the following requirements of a hypervisor for efficient virtualization [25]:

- 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 performance degradation.

Keeping in view the above requirements, different types of hypervisors have been introduced in market which are categorized into two main classes as shown in Figure 2.2

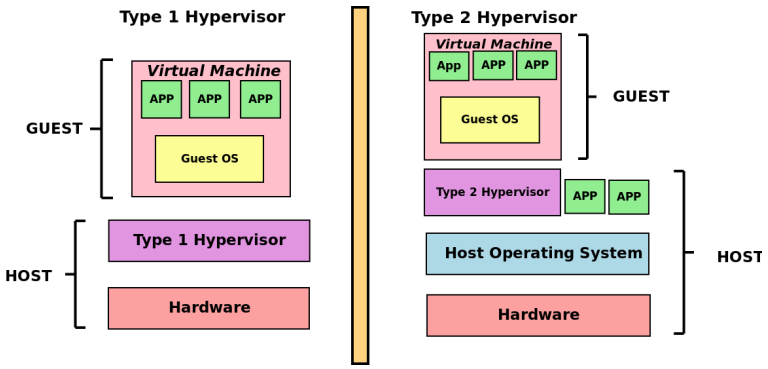

Figure 2.2.: Type-1 vs Type-2 Hypervisor

2.1.4. Types of Hypervisors

Following are two types of hypervisors:

- **Type-1 or bare-metal** 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 KVM* [24].

- **Type-2 or hosted** hypervisor runs on host operating system to manage virtual machines where 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 should be used ,mainly on client systems, where efficiency is less critical or on machines where support for a broad range of I/O devices is important and can be provided by the host operating system [26].
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 [23].

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, having one-to-one relationships with hardware, towards developing

virtualized machines running on a single platform. Talking about big players in this industry, VMware is a dominating contender. Other key players include Citrix XenServer, Microsoft Hyper-V, 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 [27]. In 2011, Younge, Andrew J., et al. [28] provided a comparison of main virtualization solutions as illustrated 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. Taken 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 i.e. low memory and efficient power usage, long battery life, smaller footprint, lower weight with reduced cost and reliability [29]. According to Steve Heath, author of book 'Embedded System Designs' [30]:

'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 distinctive capabilities of embedded systems leading to wide spread use of microprocessors today. Some of the main features are given below:

- **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 short, embedded systems are designed for performing a fixed or a few number of dedicated functions. For running other types of general applications, general purpose computers can be used. General purpose computers are also re-programmable by end users while embedded systems are not. As far as speed and size are concerned, embedded systems are smaller in size and run at fixed optimized speed while general purpose computers are bigger in size with mostly predictable speeds.

However, with the advances in hardware technology and virtualization, embedded devices can now 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 still bring advantages, by increasing their safety, reliability and security [31].

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 [32]. 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. However, a recent trend seen today is to use these devices in virtualized systems. One of the reasons is that users 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 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 systems.

2.2.2. ARM Embedded Platforms

Since the target processor used in this thesis was ARMv8 Cortex-A53, this section will provide a brief introduction on ARM, in general, and ARMv8 architectures, in particular.

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 [33]. ARM has the following RISC architectural features [34]:

- 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. ARM processors are basically used to achieve high-performance at lower cost with efficient power consumption. There is a concept of **profiles** in ARM architecture describing different versions to be used in multiple market segments [34]. Table 2.1 highlights different profiles of ARM architecture.

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

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 permissions to use protected resources. Each processor mode uses 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]:

- 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.

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

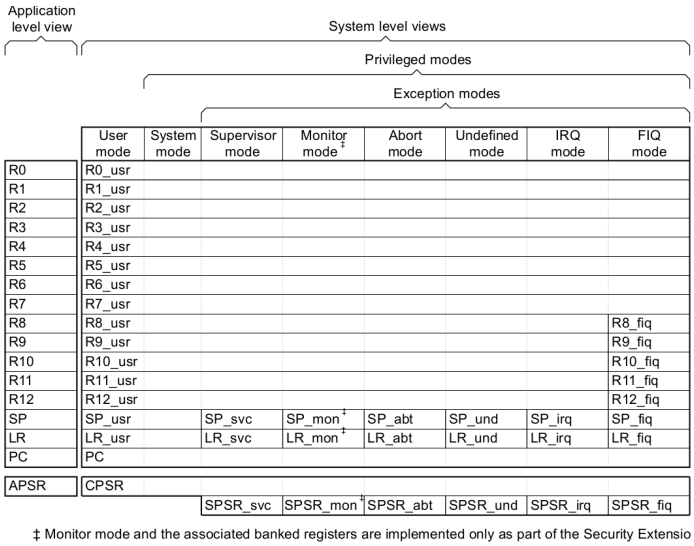

Figure 2.4.: Organization of ARM registers for different processor modes. Taken from [1]

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 [35]. In AArch64 state, the instructions can access both the 64-bit and 32-bit registers. However, in AArch32 state, the instructions can only access the 32-bit registers, and not the 64-bit registers [36]. 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. Cortex-A53 processor is a mid-range, low-power processor which implements the

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 architecture with Generic Interrupt Controller (GIC) v4 and ARM Generic Timer [37].

ARMv8-A Memory Management

Memory management unit (MMU) is a hardware that performs virtual address to physical address mapping. It does this by controlling the walk and access of 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 [37]. 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.

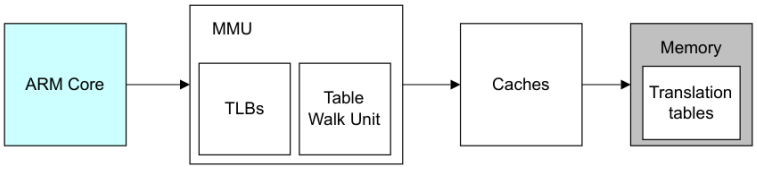

Figure 2.5.: Sructure of ARM MMU. Taken from [2]

Lets see what type of addressing scheme is used by ARMv8 platform in this thesis. The target hardware is using 48-bit virtual addresses with 4KB page size and four level of address lookup. 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 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 by MMU in Cortex-A53 processor.

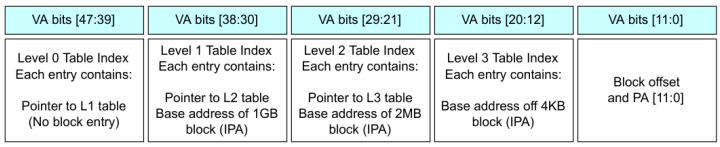

Figure 2.6.: 48-bit address translation lookup for 4KB granule size on ARMv8 architecture. Taken from [3]

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 executing multiple guests in non-secure EL1 state. Each guest can run user applications in EL0 state which is also non-secure. Address translation occurs in two stages for virtualized guests. Stage 1 translation converts virtual addresses to intermediate physical address (IPA) which is managed by virtual machine 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 OSes. ARM virtualization extensions have made generic timers (GT) and the generic interrupt controller (GIC) 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.

There are two ways of I/O virtualization. First method involves rewriting of device drivers in VMM as shown in Figure 2.7. It provides high performance but it increases the chances of introducing bugs in hypervisor code and has high engineering costs. Other method is to use para-virtualized device drivers in guest OS as shown in Figure 2.8. It has lower engineering costs and more fault tolerance but it has performance overheads.

With ARM virtualization extensions, virtualization of I/O devices 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, the provision of a System MMU (SMMU) to aid memory management, two levels of address translation and support of hardware acceleration and abstraction [38].
Xen on ARM, has taken benefit from hardware virtualization extensions and PV I/O drivers. It implements a minimal amount of hardware support directly in hypervisor

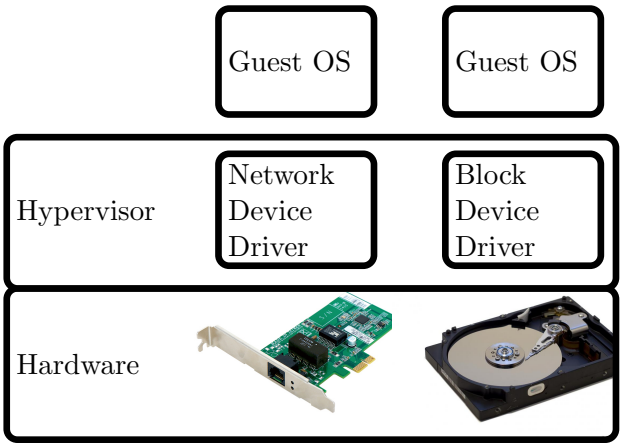

Figure 2.7.: I/O Virtualization in Hypervisor (VMM)

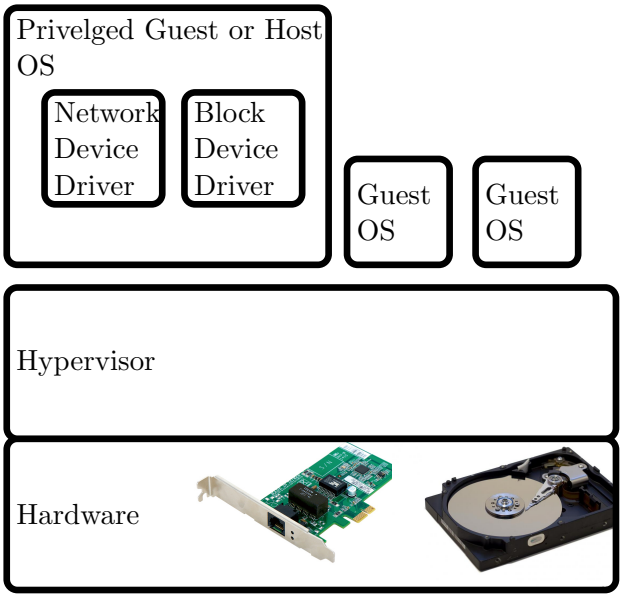

Figure 2.8.: I/O Virtualization in privileged Guest VM or Host OS

and allows a special privileged guest called Domain-0 to perform I/O using PV drivers on behalf of other non-privileged guests [39]. ARMv8 architecture provides hardware support for following virtualization:

- **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 GIC to allow hypervisor to inject virtual interrupts to guests OSes 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

Previous sections in this chapter gave an introduction to the basics of virtualization and ARM embedded platforms. The current section will present the PHIDIAS hypervisor, which will serve as the basis for porting Xen I/O architecture in this thesis.

PHIDIAS 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 after Perikles which is being integrated in industrial automotive products at OpenSynergy [40].

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 used in safety critical applications, static analysis techniques are highly recommended. 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 hence successfully used in safety critical real time applications.

2.3.2. Core Components of PHIDIAS

In PHIDIAS, memory requirements of guests are defined at compile time. It causes requested memory allocations and alignment constraints to be verified and satisfied statically. It assigns fixed physical addresses to each of those static allocations. All desired page tables, list of mapped memory ranges available to software and memory areas are finally compiled into a resultant bootable image. Structure of virtual CPU interface has been defined for simulating full privileged and unprivileged register banks.This interface enables running unmodified guests with hypervisor responsible for trapping and emulating certain sensitive instructions.

PHIDIAS hypervisor is instantiated on each physical CPU in multicore architecture running guest VMs on individual cores. Scheduler is based on a simple single-priority round-robin policy. Dispatching of interrupts to VMs is done by implementing a static interrupt dispatch table in read-only memory, whereas 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. It preallocates timer events for signaling the end of time slice for virtual CPUs and timer events for each emulated timer device. Support of basic inter VM communication is also implemented using shared memory and signaling mechanism. This signaling mechanism is based on concept of capabilities which 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, para-virtualized execution of virtual CPUs needs Virtual Translation Lookaside Buffer (VTLB). VTLB 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

All mandatory components which are required for PHIDIAS to function correctly are shown in Figure 2.9.

Following is a brief description for each of basic components.

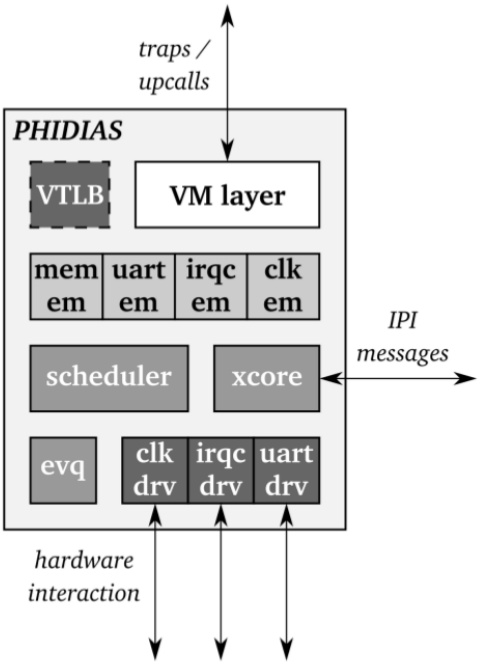

Figure 2.9.: Phidias Structure. Taken from [4]

VM Layer

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

Emulation Layer

Emulation layer consists of minimal implementation of required drivers. 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 getting zeros 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 component 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.

2.3.4. Static Configuration Framework and Final Image

Final executable image 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 in scenario file i.e. Hypervisor configuration and VM specific configuration.

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 chapter will provide a background information on the Xen hypervisor and will describe its core components of split device model for I/O virtualization.

3.1. Introduction of Xen

The Xen Project hypervisor is an open-source type-1 or bare-metal 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) [41]. It is one of the most popular open-source hypervisors which can provide both para-virtualization and full virtualization solutions. It has been used for 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: Amazon Web Services, AMD, Bromium, Calxeda, CA Technologies, Cisco, Citrix, Google, Intel, Oracle, Samsung and Verizon [42].

Xen consists of three main components which work together to provide virtualization solution. These 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 access hardware and perform I/O via Dom0. Dom0 is also responsible for launching and managing DomUs with the help of control stack called **Xen Toolstack** running on it. Figure 3.1 shows the architecture of Xen virtual environment.

3.1.1. Modes of Virtualization in Xen

Xen provides two types of virtualization modes to guests:

- **Para-virtualization (PV)**, first introduced by Xen, is a virtualization technique

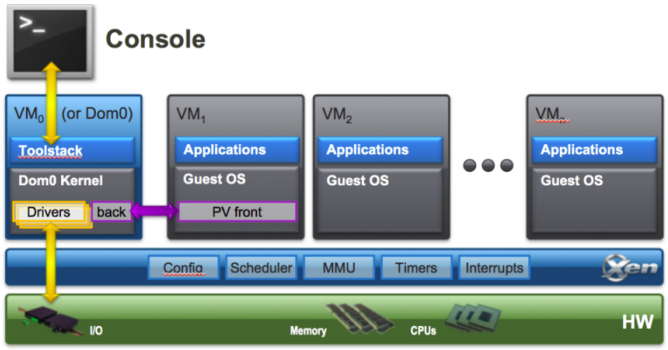

Figure 3.1.: Basic architecture of Xen virtualization. Taken from [5]

in which guests are modified and aware of being run on a hypervisor. A PV guest requires PV-enabled kernel with PV drivers to run on top of VMM. 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 hardware virtualization extensions e.g. Intel VT or AMD-V hardware extensions support 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, PV drivers can be used for I/O on HVM guests to increase performance of system.

Guests having different modes virtualization can be run at the same time on Xen. It is also possible to use para-virtualization on HVM guests and vice versa. This mixing of modes introduces two other modes of virtualization on Xen which are PVHVM and PVH. In PVHVM guests, optimized PV drivers are used for disk and network virtualization on hosts with hardware support of virtualization. PVH are basically PV guests which use PV drivers for boot and I/O and use hardware virtualization for others.

Each mode has its pros and cons. 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 [43].

On the other hand, PV mode allows modified guests to run on a system providing an abstraction of similar physical hardware to guests and provides near native performance as compared to HVM guests. It replaces the sensitive instructions with hypercalls to VMM. It also allows combining several hypercalls into one hypercall to reduce transitions between Guests OS and VMM [43].

3.2. Xen on ARM I/O virtualization

Xen on ARM 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. Dom0 is then responsible for MMIO remapping and interrupt handling.

I/O virtualization in Xen consists of two pairs of PV drivers called **frontend and backend drivers**.For each class of hardware devices i.e. disk, network, console, framebuffer, mouse, keyboard, etc, a pair of PV drivers is present in system.

3.2.1. PV Backends and Frontends

PV backends are implemented in Dom0 with their corresponding frontends are present in DomU. These PV 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 layer between frontend and backend drivers in Dom0 [6]. Figure **??** shows the general I/O device virtualization architecture in Xen.

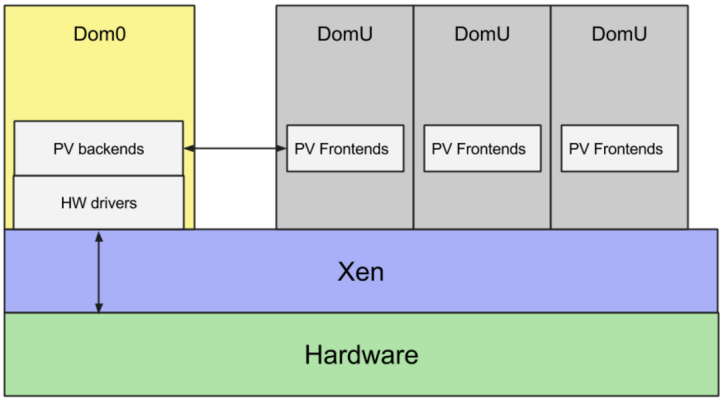

Figure 3.2.: Xen I/O Device Virtualization Architecture. Taken from [6]

3.2.2. Driver Domains

In order to provide isolation, security and disaggregation, Xen has introduced the concept of driver domains. These are unprivileged guests running backends and native I/O drivers. Driver domain does not affect other guests or domains in case of being compromised or crashed. Figure 3.3 shows the architecture of driver domain in Xen.

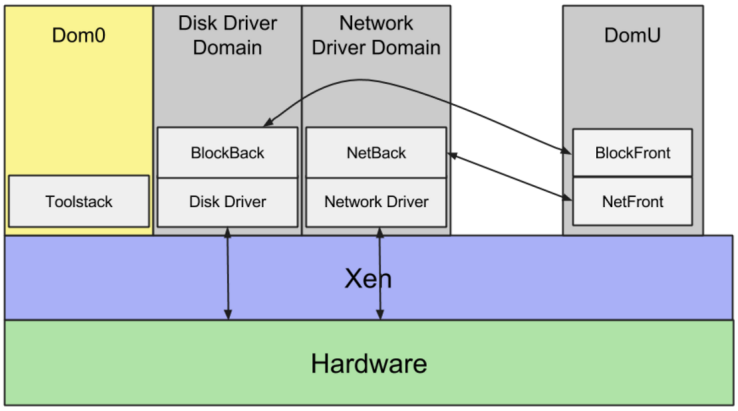

Figure 3.3.: Xen Driver Domains Architecture. Taken from [6]

3.2.3. Basic Features

Xen on ARM is a simple, smaller and faster as compared to its x86 counterpart. Some of its main features 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 PV pair of drivers for I/O virtualization.
- It runs entirely in hypervisor mode and provides hypercalls to guest VMs for switching between hypervisor and kernel modes.
- It uses virtualization aware GIC for interrupt handling.
- It uses virtualization aware GT for timer virtualization.
- It supports only one type of guests which exploits hardware virtualization as much as possible and uses PV interfaces for I/O device virtualization.

Figure 3.4 illustrates the simple architecture of Xen on ARM platforms.

3.3. Xen Split I/O Driver Model

With the prevalence of embedded systems in our everyday life, enhancements in I/O devices has increased significantly. As described in previous sections, Xen delegates all I/O devices to Dom0 or driver domains. All other guests access those I/O devices via Dom0 or driver domains. Xen uses a split driver model for multiplexing and accessing I/O devices. It supports virtualization of wide range of diverse I/O devices i.e. net, block, console, keyboard, mouse, framebuffer, XenGT(intel Graphic card), 9pfs, PVCalls, multi-touch, sound and sisplay devices [44].

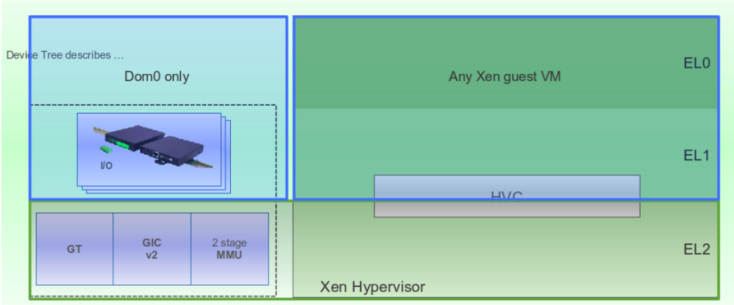

Figure 3.4.: Basic architecture of Xen on ARM platforms. Taken from [6]

3.3.1. Basic Components of Xen Split I/O Driver Model

Xen split driver model 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 native drivers for I/O devices are present in Dom0 or driver domains for accessing actual hardware. They are interfaced with backend drivers which provide a generic interface and I/O device multiplexing functionality. Frontend drivers are present in unprivileged guests and communicate with backends using shared 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 frontends and backends depends upon Xen grant tables, event channels, Xenbus and Xenstore. These mechanisms will be explained in the later sections. Figure 3.5 shows the basic communication mechanism of Xen split driver model.

3.3.2. Xen's Mechanisms for I/O virtualization

Xen provides following four basic mechanisms which are used by its split driver model:

- Grant Tables
- Event Channels
- Xenstore
- XenBus Protocol

Split drivers across domains use these mechanisms to communicate with each other and access hardware devices. A brief overview for each of the above mechanisms is provided in following sections.

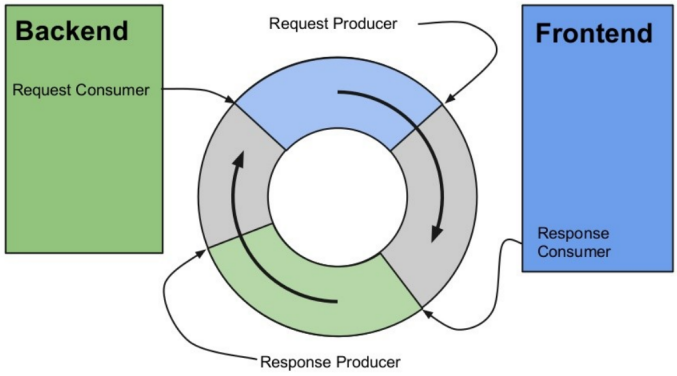

Figure 3.5.: Communication in Xen Split Driver model. Taken from [7]

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 operations to be performed on shared memory i.e. memory copying and memory transferring. Since Xen has access to all physical memory, it can easily copying the data between domains. Besides copying, it can also transfer a page from one domain to another by changing the page owner and modifying guests' page tables.

Each guest has its grant table shared with Xen. Grant table is an array of grant entries which consist of domain ID, access flags and page frame numbers of shared pages. Each entry can be indexed using grant references which are basically integer numbers. These grant references are later placed in Xenstore ,a virtual file system for device discovery in Xen, to communicate shared page information to other domains.

Xen implements hypercalls for grant table operations GNTTAB_*. Few important hypercalls are GNTTABOP_copy, GNTTABOP_transfer and GNTTABOP_map_grant_ref. Out of these, GNTTABOP_transfer is rarely used due to frequent TLB invalidations. It is used to transfer the ownership of page to another domain and leave the calling domain's address space. NTTABOP_map_grant_ref is used to map a page into current domain's memory to allow it to perform read or write operations on it. Mapping a page removes original reference of page in sender domain's address space. GNTTABOP_copy is the most common operation which duplicates the entire page into the local memory of calling domain. All split drivers heavily depends upon this operation.

Xen hypervisor creates four types of structures to implement grant 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.

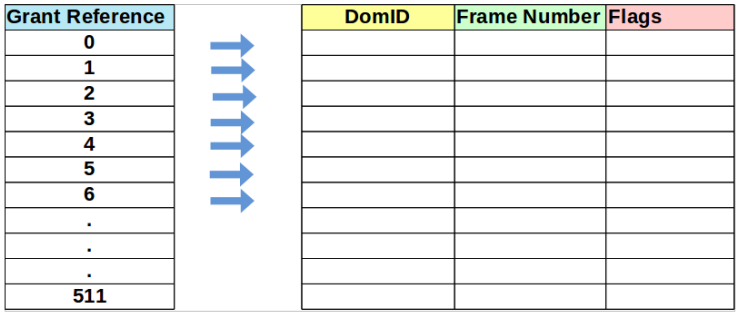

Figure 3.6.: Basic structure of Shared grant table

Grant Table Use for Shared Ring Buffers : Device I/O rings which are used for communication between split drivers are built on top this grant table mechanism. Frontend driver creates ring buffer pages and grants access to backend. Backend driver maps these shared pages for ring buffers for establishing a communication channel between domains. Figure 3.7 shows the actions performed by split drivers on ring buffers for inter-domain communication.

Xen Event Channels

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

- Inter-domain 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 (PIRQ) 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.

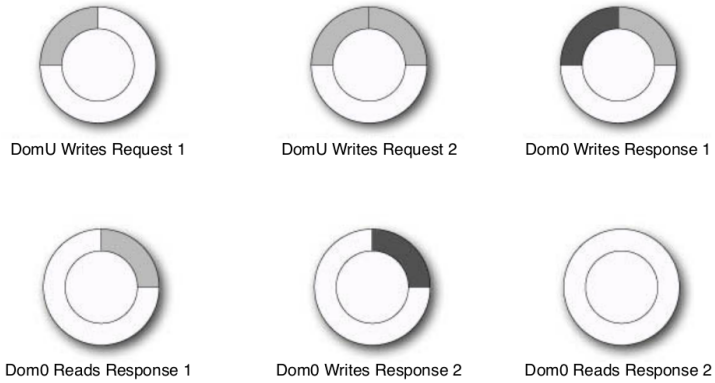

Figure 3.7.: Request/Response sequence on ring buffers. Taken from [8]

- Virtual IRQs (VIRQ) are used to bind IRQs of virtual devices e.g virtual timer or emergency console to event channels.
- Intra-domain events are used to send events between virtual CPUs of a single domain similar to interprocessor interrupt (IPI) mechanism. These are also bi-directional similar to inter-domain events.

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. 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 route events to destined remote domain.

Event channel bitmap is a structure implemented for each virtual CPU of guests running on Xen. This bitmap is shared with Xen through **'shared info'** page created during initialization of guest. When one domain sends an event to another, Xen hypervisor sets an event channel upcall pending flag and desired bit of event in destination guest's event bitmap, present in its shared info page. There are also mask bits for disabling event delivery to guests. Event delivery can be masked both by individual event type or disabling/enabling all together.

Xen supports two designs of implementation for event channels ABI [45]:

- 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 bits in event channel bitmap. For the thesis, 2-level event channel has used since 1024 events are more than enough for porting Xen's I/O split drivers to PHIDIAS.

- FIFO-based event channel ABI has lock-less 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 **??** shows the basic structures used for 2-level event channel implementation.

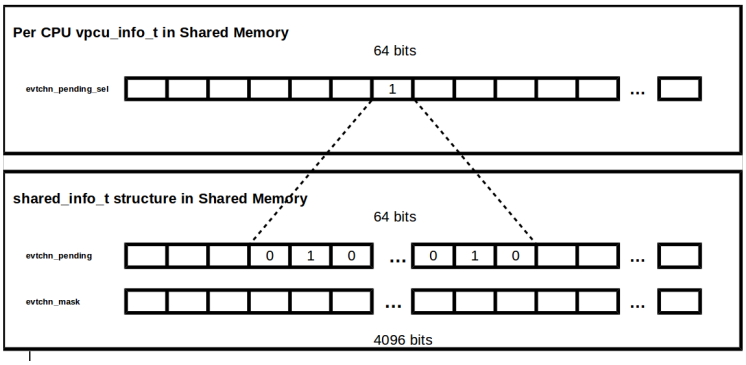

Figure 3.8.: Bitmap of 2-level event channel ABI in Xen. Taken from [9]

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 event 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.

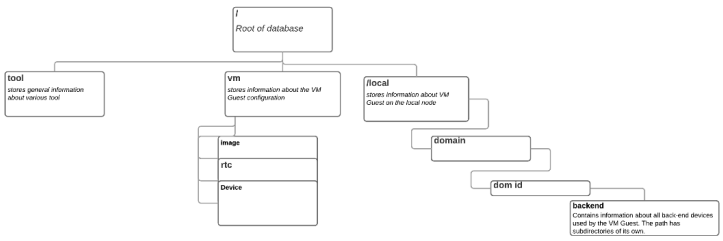

Figure 3.9.: Basic hierarchy of xenstore filesystem

There are two rings created per domain and shared with Xenstore. 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. 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 [46] 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.

Xenstore is running as a daemon called Xenstored and allows Dom0 and guests to access information about configuration and status of the system [47]. 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 [48].

XenBus

Xenbus [49] 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 PV drivers to connect VM machines with each other. Its functionality depends upon grant tables, event channels and Xenstore. XenBus implements a state machine which keeps track of several states of PV 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 related backend drivers and Xenstore are up and running.

The core component of Xenbus interface is state enumeration which both halves of split drivers should implement. This means that while registering with XenBus, PV drivers should provide with their own implementation of ***otherend_changed*** function which switches states of the driver and performs desired setup functions depending upon the changes in state of other end of driver. Xenbus interface provides API 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.

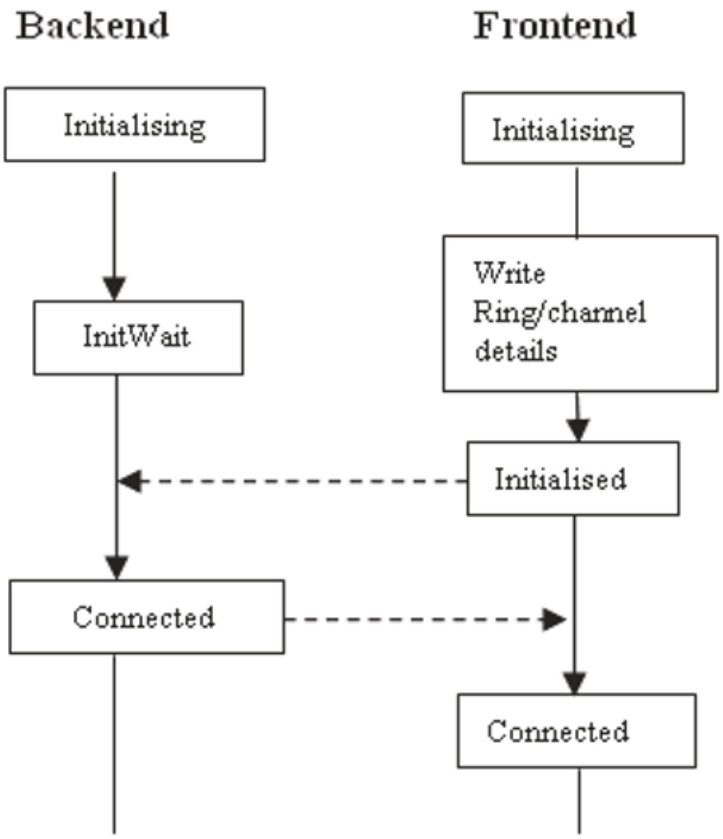

Figure 3.10.: States transition of frontend and backend drivers implemented by XenBus. Taken from [10]

4. Experimental Setup

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

4.1. Equipment for Porting

Basic equipment used in this thesis consisted of an ARM target platform and a Linux PC with required tools and Linux source code. The details of target platform and Linux kernel used are given below.

4.1.1. Target Platform

HiKey (LeMaker version) [50] was used as a target platform. Its main features are:

- 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 [51]. Linux kernel used for running PHIDIAS was linux-4.11-rc2 obtained from [52].

4.1.3. Xen version

Xen 4.7.0 was used for setting up reference framework for porting downloaded from website [53].

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. The link of patch for adding this functionality is provided in section C of Appendix A.

- Keep the changes as minimum as possible in Xen PV-enabled split drivers. General purpose interfaces should be implemented for the required changes, exposed to all Xen's split drivers. Current work had satisfied this requirement by adding changes usable by I/O split drivers.

- Add required changes in Linux kernel code cleanly which should be easy to maintain in future releases. Two major changes were added for porting work. One was to create a separate memory ZONE name ZONE_XEN in Linux kernel. It was used by split drivers for allocating pages for sharing. Another change was to implement a mechanism for registering customized IPI handler for software generated interrupts.

- Port Xen's frontend and backend of network virtual device to PHIDIAS for the proof-of-concept of running it on a static hypervisor and adding networking support between guests.

- Same tools (Iperf and Ping) should be used for testing on Xen and PHIDIAS. 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 has dependencies on Xen hypervisor. Hence it was not used in our porting setup.
- At least two guests should be able to communicate via ported network split driver.

5. Design and Implementation: Porting Xen Split Driver Model to PHIDIAS

For porting Xen split driver model, three major components i.e. Grant tables, Event channels and Xenstore are modified to use static memory and PHIDIAS's xcore and capability feature. In this chapter, I will explain how PHIDIAS's ***Principle of Staticity*** could be applied to to Xen I/O drivers.

5.1. Porting Xen I/O Virtualization Framework to PHIDIAS

All necessary porting done in core components of Xen I/O virtualization framework will be discussed in this section.

5.1.1. Porting of 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 zeroed 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. Size of this static memory range is configurable and depends 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 config option **CONFIG_XEN_DOM_ID** as shown in listing 5.1.

```
Shared_info_pages = (unsigned long)xen_remap(0xfee43000, XEN_PAGE_SIZE * 2)
    ;
if (!Shared_info_pages) {
pr_err("not enough memory\n");
return -ENOMEM;
}

if (xen_initial_domain())
memset_io((void *)Shared_info_pages, 0, XEN_PAGE_SIZE * 2);

HYPERVISOR_shared_info = (struct shared_info *)(Shared_info_pages + (
    XEN_PAGE_SIZE * CONFIG_XEN_DOM_ID));
```

Listing 5.1: Guest mapping its shared information page on PHIDIAS

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.

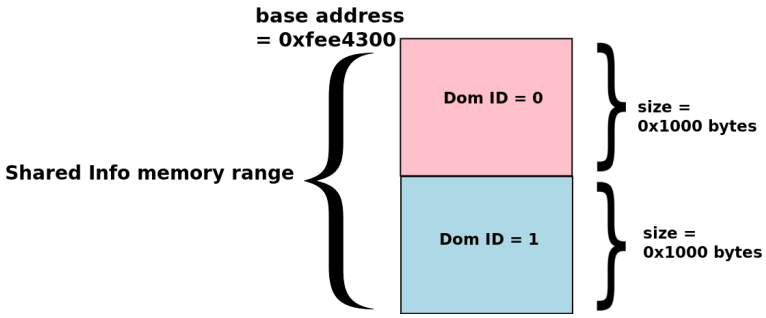

Figure 5.1.: Basic approach of implementation of shared info pages in PHIDIAS for two guests

5.2. Porting of Grant Tables

As described in 3.3.2, grant tables are a mechanism for sharing memory pages across domains. Xen hypervisor allocates pages for grant tables per domain and guests map them into their address spaces. By design, Xen does not support swapping which means that even unused part from the allocated memory of a guest could not be used by other guests. To solve this problem, Xen uses **ballooning**. Ballooning is way of dynamically increasing or decreasing size of allocated memory. 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 unused blocks of memory to hypervisor. However, guest can not retrieve more memory from hypervisor's memory pool through ballooning than its maximum configured amount specified during its startup. Memory allocated to unprivileged guests is taken from Dom0 memory pool and hence 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's PV split drivers, ballooning is used to allocate pages for grant tables, XenBus ring buffers and mapping received network packets in network backend driver from the related frontend. In our ported setup, it has been replaced by defining two static globally shared readable/writable memory ranges for PHIDIAS's 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. Guests allocates

required pages from these two memory ranges and map them un-cached into their respective address spaces. These two memory ranges will be explained in more detail in the following sections.

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 had used 32 pages for its own grant table implementation. In our setup, every guest had access to other guests' grant table through un-cached mapping of their grant pages into its address space. For performing grant table operations, I/O split drivers of Xen find a reference or index of unused entry in their guest's grant tables and update this entry with fields of corresponding 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 setup, hypercalls are used to perform grant operations i.e. granting access to remote domain, mapping, transferring and copying pages across domains. In our ported setup, since each guest had access to remote's dguest's grant tables by mapping them directly into its address space, a local guest could perform direct operations by manipulating the fields in grant table entries of a remote guest. Listing 5.2 shows how a guest mapped remote guest's grant table into its domain in our setup. Figure 5.2 shows the configuration of grant table static memory range for two guests in PHIDIAS.

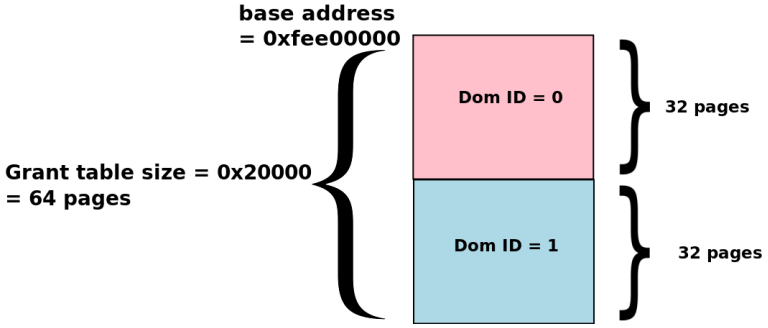

Figure 5.2.: Static memory configuration for grant table in PHIDIAS for two guests

```
static int gnttab_setup(void)
{ ...
        gnttab_shared.addr = xen_auto_xlat_grant_frames.vaddr;

        if(xen_initial_domain())
            remote_grant_frames = 0xfee00000 + (0x20000 * 1) ;
        else
            remote_grant_frames = 0xfee00000 + (0x20000 * 0) ;

        vaddr = xen_remap(remote_grant_frames, XEN_PAGE_SIZE *
            max_nr_gframes);
```

```
        gnttab_shared_remote.addr = vaddr;
...
}
```

Listing 5.2: Code snippet for mapping remote guest's grant table in case of two guests configured for the ported setup

5.2.2. Static Memory for I/O Split Drivers Usage

I/O Split drivers of Xen guests play with memory using a set of in-built kernel functions e.g alloc_page, free_page etc. However, they share those pages with other guests through Xen's 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 for our setup. A static globally-shared memory range of size 8192 KB has been configured in PHIDIAS for two guests. Size of ZONE_XEN for each guest was set to 4096 KB i.e. 1024 pages. Each guest obtained the starting address of its zone using the base address of configured shared memory range of size 8192 KB and its domain ID as shown in listing 5.3.

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

Listing 5.3: 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. Since the entire zone memory range is shared between two guests in our

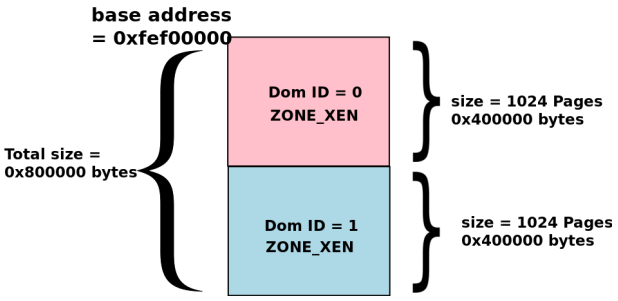

Figure 5.3.: Static memory configuration for ZONE_XEN in PHIDIAS for two guests

setup, each guest could easily calculate the starting address of other guest's zone space. In order to access pages granted for sharing by a remote's guest, the local guest should map those pages into its address space so that it could read or write on those shared pages. In native Xen setup, Xen hypervisor is responsible for modifying page tables and sharing of pages across domains. However, in our setup, we have to map all pages of other's guest ZONE_XEN into local guest address space before accessing them through I/O split drivers. We cannot use **ioremap** function when grant operations are called

by split I/O driver e.g in __gnttab_map_grant_ref and gnttab_batch_copy. The reason is that ioremap cannot be called in interrupt context and PV split drivers perform grant operations in this particular context. In order to solve this problem, whole zone area of other guest had been mapped in local guest's address space during initialization in arch_gnttab_init function as shown in listing 5.4.

```
int arch_gnttab_init(unsigned long nr_shared)
{ ...

#if CONFIG_XEN_DOM_ID == 1
base_value = 1044224;
phys_addr = 0xfef00000;

#elif CONFIG_XEN_DOM_ID == 0
base_value = 1045248;
phys_addr = 0xFF300000;
#endif


virt_addr =  xen_remap( phys_addr , 0x400000);
for (i = 0; i< SIZE_ARRAY; i++)
{
key = phys_addr >> XEN_PAGE_SHIFT;
hash_insert(key, virt_addr);
virt_addr = (unsigned long)(virt_addr) + 4096;
phys_addr = phys_addr + 4096;
}
..
}
```

Listing 5.4: Code snippet for populating hash table for virtual address mapping of shared pages for two guests in ported setup

Then a hash table had been created containing virtual addresses of these mapped pages indexed with their respective physical page frame numbers. This hash table had thus solved the following two problems:

- Speed up the process of finding virtual address of remote's guest shared page by not mapping each page individually in grant table functions.
- Avoiding the use of ioremap in grant table functions since it cannot be called in interrupt context.

Figure 5.4 shows the basic structure of implemented hash table.

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) can be used for

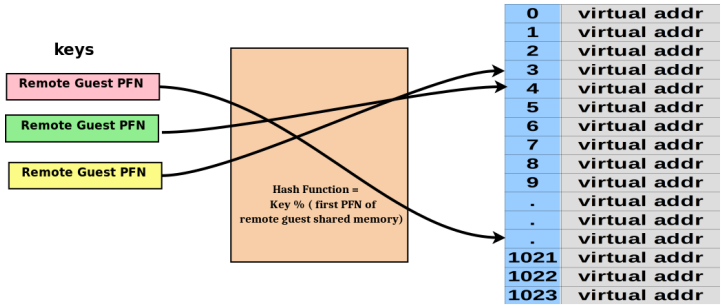

Figure 5.4.: Structure of hash table used for accessing remote guest's of ZONE_XEN shared pages

interprocessor communication. There are 16 SGI available on ARM architecture. For 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 which is configured in hypervisor node of its flattened device tree provided to it by Xen during its booting sequence as shown in listing 5.5.

```
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 5.5: Xen Hypervisor node in hi6220 flattened device tree

In PHIDIAS, SGIs have been used for implementing interprocessor interrupts using its **Xcore mechanism**. By default, first 6 SGI interrupts are used by Linux kernel. In our ported setup, we could use one of the remaining SGIs for registering **xen_arm_callback** handler for Xen event IRQ. However, in Linux kernel version 4.11_rc2 used in this work, Linux kernel handle_IPI function only processes first 6 SGIs. 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 other than the default first six, modified IPI handling function is implemented for our setup. It checked whether some handler is registered for given interrupt number and then called respective handler if found any. An online resource [54] has been used as a reference for implementing this IPI handling

function.

I had used SGI number 9 for triggering Xen events in PHIDIAS for inter-domain 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 event channel hypercalls in Xen guests were replaced with PHIDIAS specific code working on shared memory and Xcore capabilities.

5.3.1. Static Memory Allocation for Event Domain Pages

Xen hypervisor maintains a structure for event channels 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 a 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 guest domain. For remote sharing of Xen event domain structures, a static globally shared memory named **event_domains** has been configured in PHIDIAS and mapped into 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 an array of event domain structures so that it could write its domain ID and local event port number while binding inter-domain event channels. Figure 5.5 shows the basic structure of event domains pages in PHIDIAS for two guests.

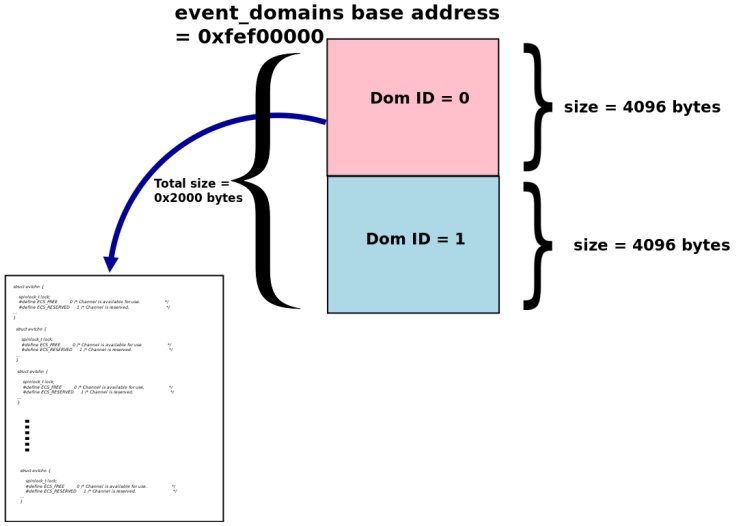

Figure 5.5.: Basic structure of shared event domains pages in PHIDIAS for two guests

5.3.2. Adding Capabilities for IPI in PHIDIAS

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

- Sending event notifications between Xenstore userspace application and PV I/O drivers, behaving more like an intra-domain event.
- Sending event notifications between frontend and backend of split drivers, behaving more like an inter-domain event.

For the above two types of events, two capabilities had been added in guest configuration on PHIDIAS as shown in listing 5.6.

```
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 5.6: 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 **inter-domain events** and second capability emulated **intra-domain events**. Both these capabilities triggered SGI 9 for Xen events in virtual guests.

5.4. Design of Porting Xenstore

As explained previously in section 3.3.2, each guest in Xen shares a page consisting of ring buffers for requests/responses with Xenstore. It also binds an event channel with Xenstore daemon to send event notifications. A ***Xen filesystem (xenfs)*** driver is used for creating and mounting files for communication between guests and Xenstore. Out of these files, following two are used for communication between Xenstore and Xen Dom0 guest:

- xsd_kva for mapping Dom0 shared page into Xenstore.
- xsd_port for getting an unbound event channel number of Dom0 used for binding it with an event port in Xenstore daemon.

In native Xen setup, Dom0 creates unprivileged guests and manage them through domain control specific hypercalls. During the process of creating DomU, a Xenstore interface page and an unbound event channel are allocated which are then introduced to Xenstore daemon through Xen's XL tool. Later during initialization, DomU maps this Xenstore interface page into its address space and gets the xenstore event channel number with

the help of hypercalls.

In our current work, since no XL tool had been ported and Dom0 did not control creation of DomU guests, DomU Xenstore interface page and event channel number were added manually in Xenstore.For this purpose, a similar file interface as xsd_kva had been exported to Xenstore daemon by Dom0 used for mapping DomU's Xenstore page as shown in listing 5.7. Two globally shared static pages were allocated for implementing Xenstore interfaces of guests in PHIDIAS as shown in Figure 5.6. For event channel of DomU, event port number 1 was hard-coded in Xenstore daemon. DomU had been introduced manually in Xenstore application in file xen/tools/xenstore/xenstored_domain.c as shown in listing 5.8.

```
...
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;

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

```
static int dom0_init(void)
{
...
domU = find_domain_by_domid(1);

if (domU == NULL)
```

```
{
/* Hang domain off "in" until we're finished. */
domU = new_domain(dom0->conn->in, 1, 1);
if (!domU) {
return -1;
}
domU->mfn = 0xfee45000 >> 12;
domU->interface = xenbus_map_foreign();
if (domU->interface == NULL) {
return -1;
}

/* Now domain belongs to its connection. */
talloc_steal(domU->conn, domU);

fire_watches(NULL, "@introduceDomain", false);
}

domain_conn_reset(domU);
...
}
```

Listing 5.8: Manual introduction of DomU in Xenstore during dom0 initialization

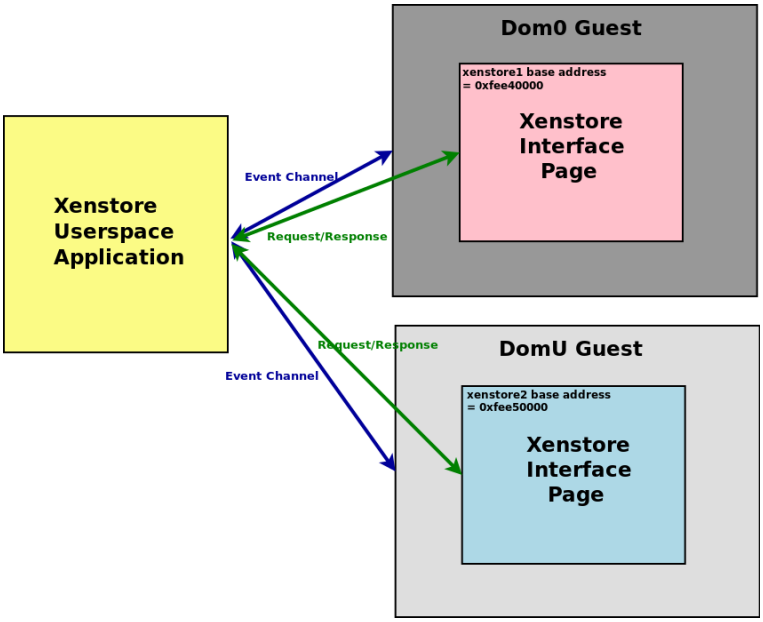

Figure 5.6.: Communication between Xenstore application and Guests in PHIDIAS

6. Porting Xen Network Virtualization

In this chapter, Xen network virtualization architecture will be explained in the first part and then the work related to porting it to PHIDIAS will be presented in second part. Other split drivers for different I/O devices can ported using the same approach adopted for PV network drivers with zero or little changes if required.

6.1. Xen Network Virtualization

This section will give an overview of Xen network's virtualization architecture and data flow. Details of bringing up virtual network interfaces in Xen will also be explained. Figure 6.1 shows basic architecture of Xen network virtualization framework.

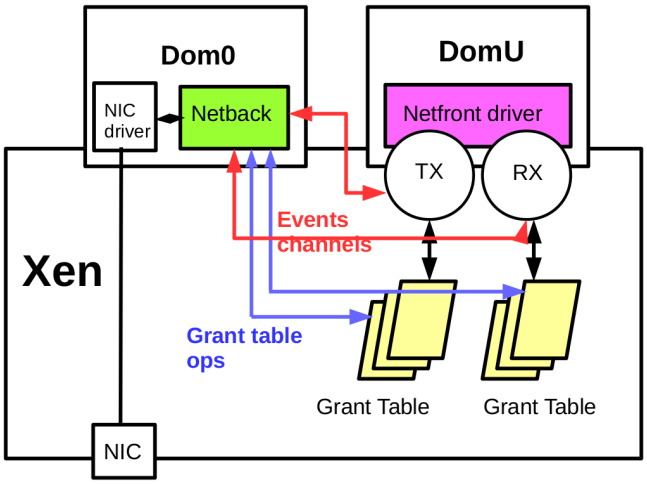

Figure 6.1.: Xen network virtualization architecture

6.1.1. Backend and Frontend drivers of XenBus

All Xen split I/O drivers depend upon a general bus entity named XenBus which provides an interface to backend and frontend drivers for establishing communication between each other. The internal working of Xenbus is dependent upon Xenstore and event channels. Xenbus, itself, is composed of two split drivers that work together to provide functionality of inter-domain communication as shown in Figure 6.2. These two halves of XenBus split driver model 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 and registering their watch points in Xenstore for getting notification about changes in the backend devices. All frontend drivers register themselves with XenBus frontend.

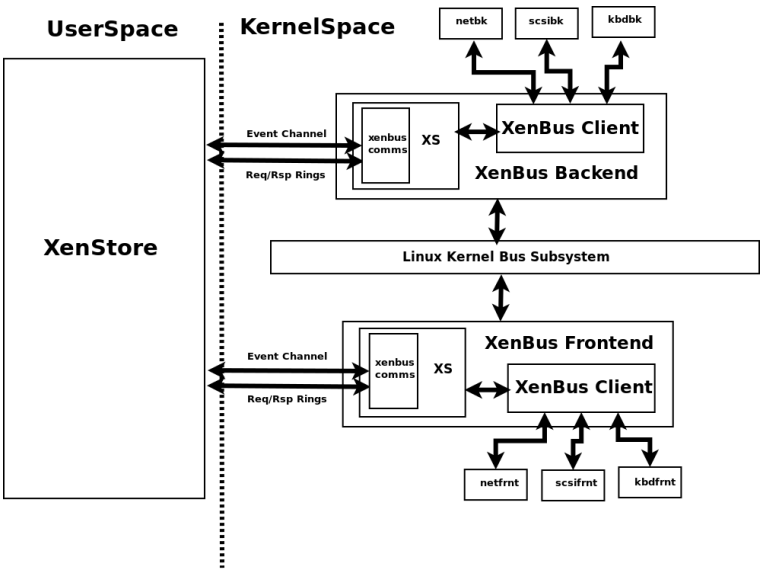

Figure 6.2.: XenBus Architecture for connecting backends with frontends in Xen

6.1.2. Netback Driver

Netback driver implements the backend in Dom0. It communicates with network frontend 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/receiving network packets to/from actual network hardware using native NIC driver.

6.1.3. Netfront Driver

Netfront driver implements the frontend in unprivileged domains. It creates Tx and Rx rings for sending instructions about data flow to the network backend driver. These rings do not contain 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

Probing of Netback Driver

In native Xen setup, netback driver is probed when Xend toolstack in Dom0 creates **'vif'** device. In the netback_probe function, network backend sets its state to XenbusStateInitialising. It then write keys to the Xenstore related to network features it supports. 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's XL configuration file for interfacing virtual network interface with real network device and kickoffs other necessary scripts to setup the bridging between virtual and real network interface. There are three main types of network setup supported by Xen i.e. bridged, routed and NAT [55]. Its default mode is bridged and from Xen 4.3 onwards, support of openvswitch is also provided.

Probing of Netfront Driver

Netfront driver gets probed by xenbus_probe function during its initialization only if Xenstore and netback are up and running. Netfront probe function 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 be notified for changing its bus state to XenbusStateInitWait. When netfront driver becomes notified about XenbusStateInitWait state 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 access permissions to netback, allocates Tx and Rx event channels and write 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 filesystem. It also allocates Rx buffers for receiving packets from netback. After completing its talk with netback, it sets its state to XenbusStateConnected.

Connection of Netfront and Netback

On receiving state change notification about successful connection from netfront, netback reads the keys written by netfront in Xenstore filesystem. It reads grant references of

Tx and Rx rings and maps them into its 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 from network stack which is basic structure used by Linux kernel to manage network packets. First, it allocates grant reference and corresponding Tx request for linear part of skb_buff. Then it calculates the number of fragments of network packet and allocates grant references and Tx requests for remaining fragmented portion 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 128 bytes for TX copy operation. If the packet size is larger than 128 bytes, remaining data is mapped into Dom0 address space. After filling out a newly allocated skb_buff with received network packet contents, 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's address space. After getting required number of slots in RX ring, it copies the packet data into those mapped Rx buffers through gnttab_batch_copy function. This in turn calls hypercall responsible for transferring data into netfront Rx buffers. Netback driver relies heavily on this batch copy operation for copying data to and from an unprivileged guest via the grant references in the RX and TX ring buffers. Netfront driver, on the other end, calls xennet_poll function to retrieve the sent network packets and forwards it to Linux network stack.

6.2. Architecture of Ported Network Virtualization Framework to PHIDIAS

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

6.2.1. Writing Keys to Xenstore for Network Virtualization

For successful initialization and inter-domain communication of I/O split drivers, Xenstore should be up and running. In case of our setup, Xenstore was cross-compiled using build root method as described in Xen wiki page [56]. 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 to provide access to the shared Xenstore pages of Dom0 and DomU to user-space. 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 access to DomU of ID 1 to write or read keys in Xenstore filesystem. 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 Dom0 to know beforehand all the keys information which DomU would need to read/write/modify in Xenstore. The simpler approach is to start Xenstore daemon with **priv-domid 1** which will enable guest with Domain ID 1 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 service of Xen toolstack .In case of our setup, since there was no Xend service, netback driver 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 in our ported setup, 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 when XL toolstack in Dom0 creates and starts DomU. In PHIDIAS, both Dom0 and DomU boots at the same time. Netfront driver should only probed after running Xenstore and netback driver in Dom0. To notify DomU guest about running Xenstore daemon in Dom0 and probing netfront 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 for this capability. In netfront driver, an IPI handler, listed in 6.4, was registered which was called on receiving an IPI 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 trigger an IPI from Dom0 from DomU, a small kernel module was implemented which triggered the capability with index 2. This module was inserted after running Xenstore daemon and writing network related Xenstore keys by Dom0 guest. Code for triggering this capability is shown in slisting 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 transmitting packets 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 Linux network stack allocates skb_buff from Linux **NORMAL** memory zone in native Xen netfront driver, it should be copied into ZONE_XEN to share it netback driver. For this, a function xen_skb_copy was implemented in skbuff.c file to copy skb_buff from NORMAL memory zone to XEN zone and called in xennet_start_xmit. Linux 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 our implementation of xen_skb_copy which is provided in listing 6.6.

```
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 snippte of function xen_skb_copy for copying entire network packet from NORMAL memory zone to XEN memory zone and its invocation from xennet_start_xmit from

The above copy operation resulted in degradation of the networking performance for large packet sizes. The resulting affects will shown for 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 to the provided host virtual address in Dom0. Actually, Dom0 allocates pages from NORMAL memory zone in its address space and provides virtual addresses of those pages as host addresses in GNTTABOP_map_grant_ref hypercall. Xen hypervisor then creates a mapping of shared pages to point to these host virtual addresses in Dom0's page tables. Since netback allocates pages for virtual host addresses from its own NORMAL memory zone, it has corresponding struct page definitions for all those pages in its mem_map array [57] and can safely use virt_to_page macro while filling fragments of skb_buff for received packets. In Linux, network stack uses this struct page definitions in functions related to dealing with fragments of network packets. The macro virt_to_page returns struct page's of given virtual address which is then used in function xenvif_fill_frags to fill skb_buff fragments via _skb_fill_page_desc function.

In our ported setup, shared pages are mapped using ioremap. The macro virt_to_page cannot be used for virtual addresses returned by ioremap. Dom0 only allocates struct page's for the memory range given to it by PHIDIAS. It has no struct page's for DomU's shared memory, otherwise its kernel's buddy allocator [58] can allocate pages from DomU's memory. In order to solve this problem, netback driver in PHIDIAS maps the shared pages containing network data and then copies data into its newly allocated pages from NORMAL memory zone as shown in Figure 6.3. It then forwards these pages to Linux network stack for further processing. Since large fragmented packets span multiple pages, this additional copy operation in our implementation had resulted in decreased network performance which will be shown later 7.

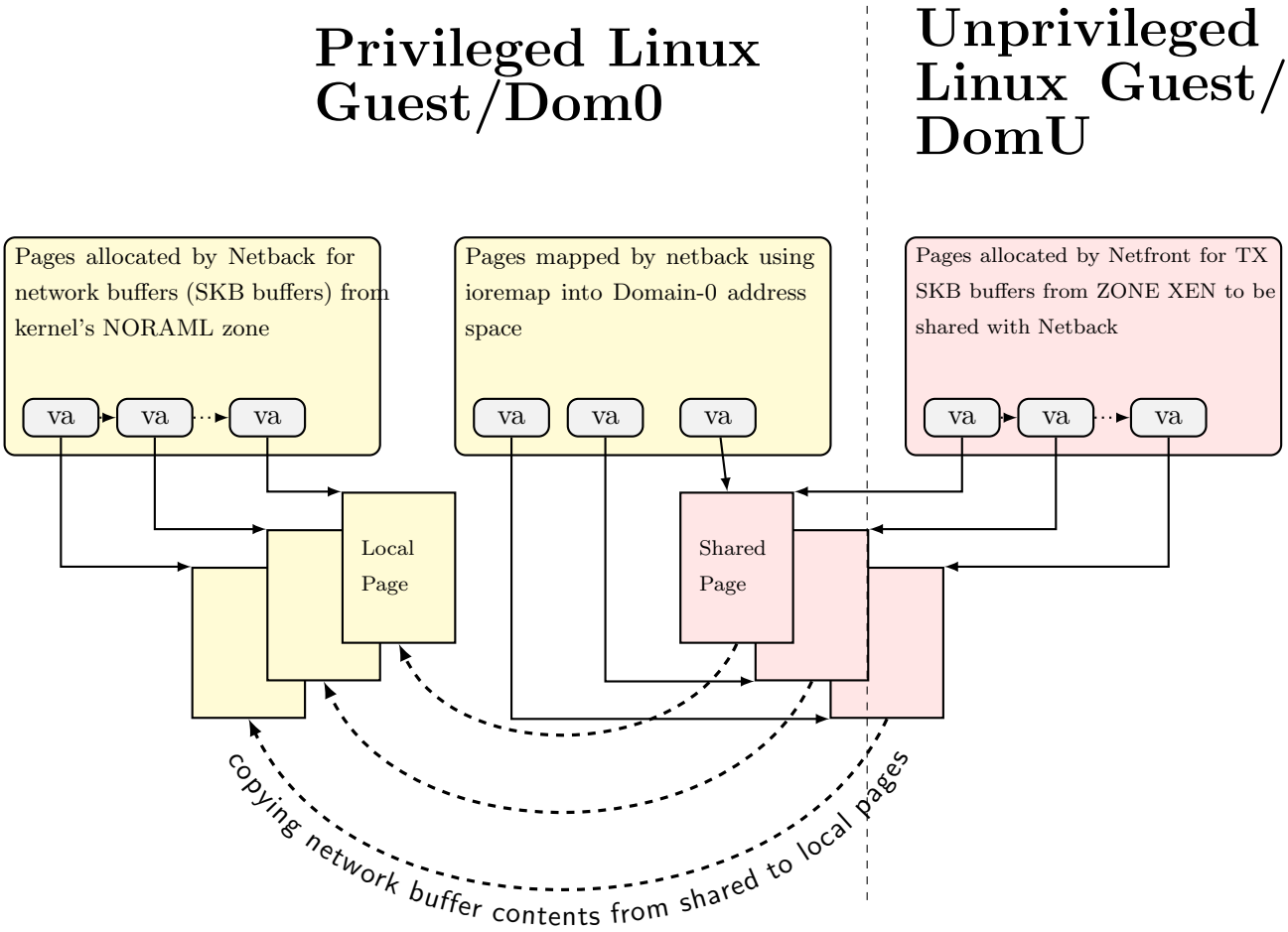

Figure 6.3.: Netback copy operation in TX path from netfront to netback on PHIDIAS

6.3. Porting other virtual I/O devices in Xen

In this section, steps require for porting remaining Xen I/O virtual devices, besides network, 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

Block frontend 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 an unbounded event channel and passes it to backend through Xenstore. It reads Xenbus state of block backend and sets it own state while establishing connection. States are changed at different steps while 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 Dom0 after successful running of Xenstore daemon and block blockend. 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 by Dom0 through a customized script. User could locate the necessary keys needed by frontend drivers by looking through their source code. For probing the frontend drivers, an IPC capability is needed to be configured in PHIDIAS and gets triggered by a simple kernel module from Dom0 to DomU.

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 types of tests were conducted for this work. One test was performed to calculate the latency of transferring network packets between Dom0 and DomU and the second one was done to measure network throughput. Both tests were conducted 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 **BusyBox v1.22.1 (Debian 1:1.22.0-9+deb8u1) multi-call binary**. It was obtained from pre-built ramdisk (initrd) image of 96board from web source [59].

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 [60]. 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

Figure 7.1 shows the comparison between round-trip latencies of ping packets on PHIDIAS and Xen. From the given results, we see that PHIDIAS outperforms Xen. This is due to the fact that no context switch and hypercalls into hypervisor were done in case of our ported network virtualization framework. However, there was packet loss in case of PHIDIAS for packets larger than MTU due to IP reassembly timeout. When fragmentation occurs for packets in our setup, two additional memcpy operations were performed as described previously in sections 6.2.4 and 6.2.5. Also since XEN ZONE pages were mapped un-cached into guests' address spaces for our ported setup 5.2.1, performing memcpy on these un-cached pages had added more time in processing of packets resulting in IP reassembly timeout for fragmented packets. Figure 7.2 shows the packets loss comparison between native Xen and PHIDIAS setup.

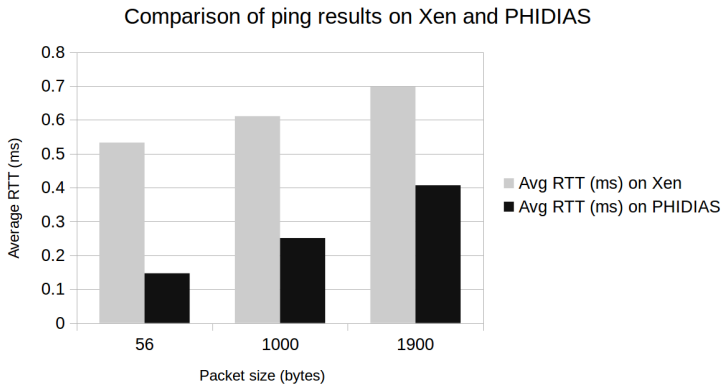

Figure 7.1.: Comparison between round trip latencies of ping packets on PHIDIAS and Xen

7.2. Network Throughput Test

Iperf utility is mostly used for measuring throughput using data streams. It is based on a client and server model. It is capable of measuring throughput between two ends in one or both directions [61].

For conducting tests, Iperf version 2.0.10 (11 Aug 2017) for pthreads [62] was used.

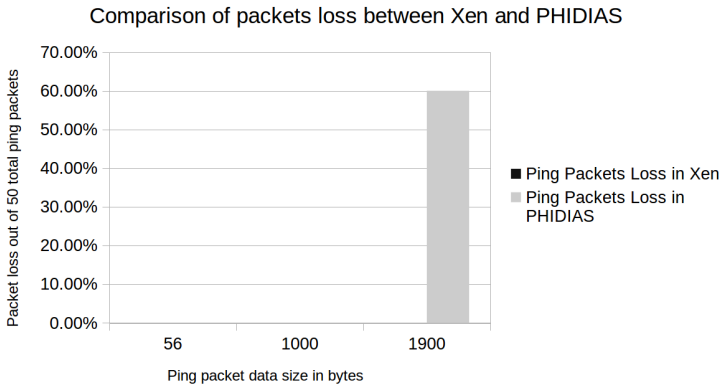

Figure 7.2.: Comparison between loss of ping packets on PHIDIAS and Xen

The tool was cross-compiled for ARM architecture. Two types were tests were conducted with reversing the roles of client and server between Dom0 and DomU guests. Each test was conducted five times in a specific context and then average was calculated for final reading. Default TCP window size of 85.3 KBytes was used for all tests and each test lasted for the default 10.0 sec interval duration.

7.2.1. Network Throughput Test results on Xen

Table 7.3 shows the results of two types of iperf tests performed between Dom0 and DomU through virtual network devices on Xen. For the first test, iperf client was run in Dom0 sending data to the iperf server in DomU and for the second test, roles were reversed.

Table 7.3.: Iperf test results on Xen

Iperf Client	Iperf Server	TCP window size on server and client	Interval duration	Transfer Bytes	Bandwidth
Dom0	DomU	85.3 KBytes (default)	10.0 sec	1.582 Gbytes	1.358 Gbits/sec
DomU	Dom0	85.3 KBytes (default)	10.0 sec	1.502 Gbytes	1.29 Gbits/sec

7.2.2. Network Throughput Test results on PHIDIAS

Table 7.4 shows the results of two types of iperf tests on PHIDIAS with first test making Dom0 as an iperf client and DomU as server and the second test with reversed roles for guests.

Table 7.4.: Iperf test results on PHIDIAS

Iperf Client	Iperf Server	TCP window size on server and client	Interval duration	Transfer Bytes	Bandwidth
Dom0	DomU	85.3 KBytes (default)	10.8 sec	429.6 Kbytes	328.6 Kbits/s
DomU	Dom0	85.3 KBytes (default)	10.0 sec	815.144 Kbytes	602.8 Kbits/s

7.2.3. Analysis of Network Throughput Test results on PHIDIAS

Figure 7.3 shows the comparison between throughput measurements on XEN and PHIDIAS for both test scenarios.

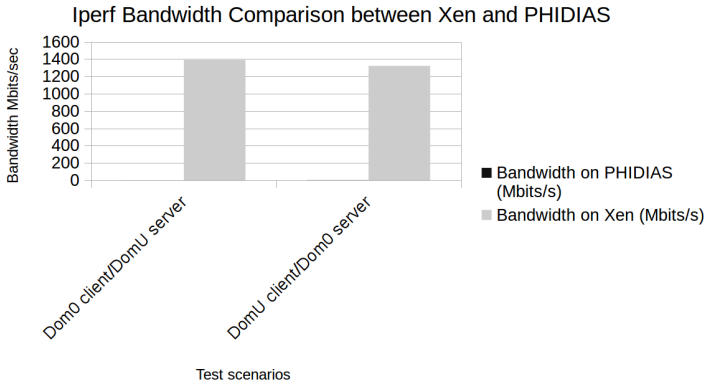

Figure 7.3.: Comparison between bandwidth measurements of Iperf on PHIDIAS and Xen

Figure 7.4 shows the comparison between data transferred on XEN and PHIDIAS for the iperf tests on XEN and PHIDIAS.

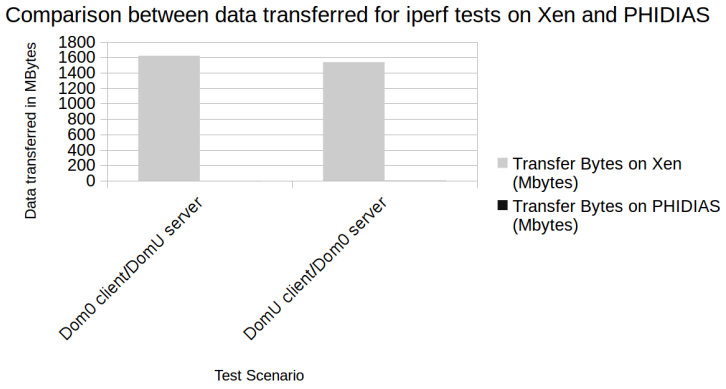

Figure 7.4.: Comparison between amount of data transferred for iperf tests on PHIDIAS and Xen

Figures 7.3 and 7.4 clearly shows a large difference in throughput measurements between XEN and PHIDIAS. Main reason for this has already been described in section 7.1.3. Iperf transfers the data depending upon the TCP window size set on both sides which is 85.3 KBytes by default. If the amount of data sent is larger than MTU, fragmentation occurs. From the results, it is obvious that iperf had caused large amount of fragmentation in TCP packets, leading to IP reassembly timeout in our setup. The time

out was due to overhead caused by two additional memcpy operations in our implementation.

8. Conclusion and Future Work

8.1. Conclusion

This thesis provides an insight into details related to porting Xen I/O virtualization framework to PHIDIAS. All necessary modifications needed for porting dynamic components of Xen split driver model to a statically configured hypervisor are discussed. This includes changes for PV I/O drivers of Xen in general and for virtual network devices in particular. Tests for comparing network performance between native Xen and our ported setup have shown that PHIDIAS outperforms Xen for packets with sizes smaller than default Ethernet MTU size. Performance degradation happens only for fragmented packets which occurs due to overhead of two additional memory copy operations on statically shared un-cached memory pages for the guests on PHIDIAS. Keeping in view the time restrictions and the goal of using a simpler approach with less changes overall, these additional memcpy operations seemed to be most appropriate implementation strategy adopted in this thesis.

This work has contributed to the extension of PHIDIAS functionality regarding support of I/O virtualization. Keeping changes minimum in both Xen's PV I/O drivers and PHIDIAS hypervisor has served two purposes. First, it has proved the flexibility of PHIDIAS static design and kept its integrity proof by symbolic execution valid. Second, it has enabled to reuse the maintenance support of Xen community.

8.2. Future work

There are still several enhancements that could be performed to expand the feature set of PHIDIAS. It includes passing-through of physical network devices (WLAN and USB-Ethernet) on HiKey ARMv8 target to domain0 in order to talk to outside world through network interface and porting Xen I/O drivers for x86 target to PHIDIAS.

The main goal of this thesis is to provide a proof-of-concept for working I/O split driver of Xen on PHIDIAS for ARM target. It has successfully achieved this purpose by connecting logical network devices on virtual machines running on top of PHIDIAS. However, lower performance for fragmented packets was obtained for throughput tests. In order to improve performance, a different technique for accessing shared network packets from netfront by netback could be used. One possible approach would be to allocate pages in local address space of Dom0 guest, equal in number to Tx requests in Tx ring buffer, and modify the corresponding page table entries of allocated pages to point towards

shared physical pages. Figure 8.1 shows the working of this technique. The reason of not implementing this approach in this thesis is that it requires implementation of new functions in Linux kernel, related to deleting the old mappings from page tables using kernel virtual addresses, modifying page table entries for those old virtual addresses to point towards shared physical pages, updating page reference counts and links in kernel and freeing and un-mapping these pages correctly after using them. This approach would hopefully be faster than our current approach as in the end, it is just changing some integers inside kernel's page tables instead of copying entires pages.

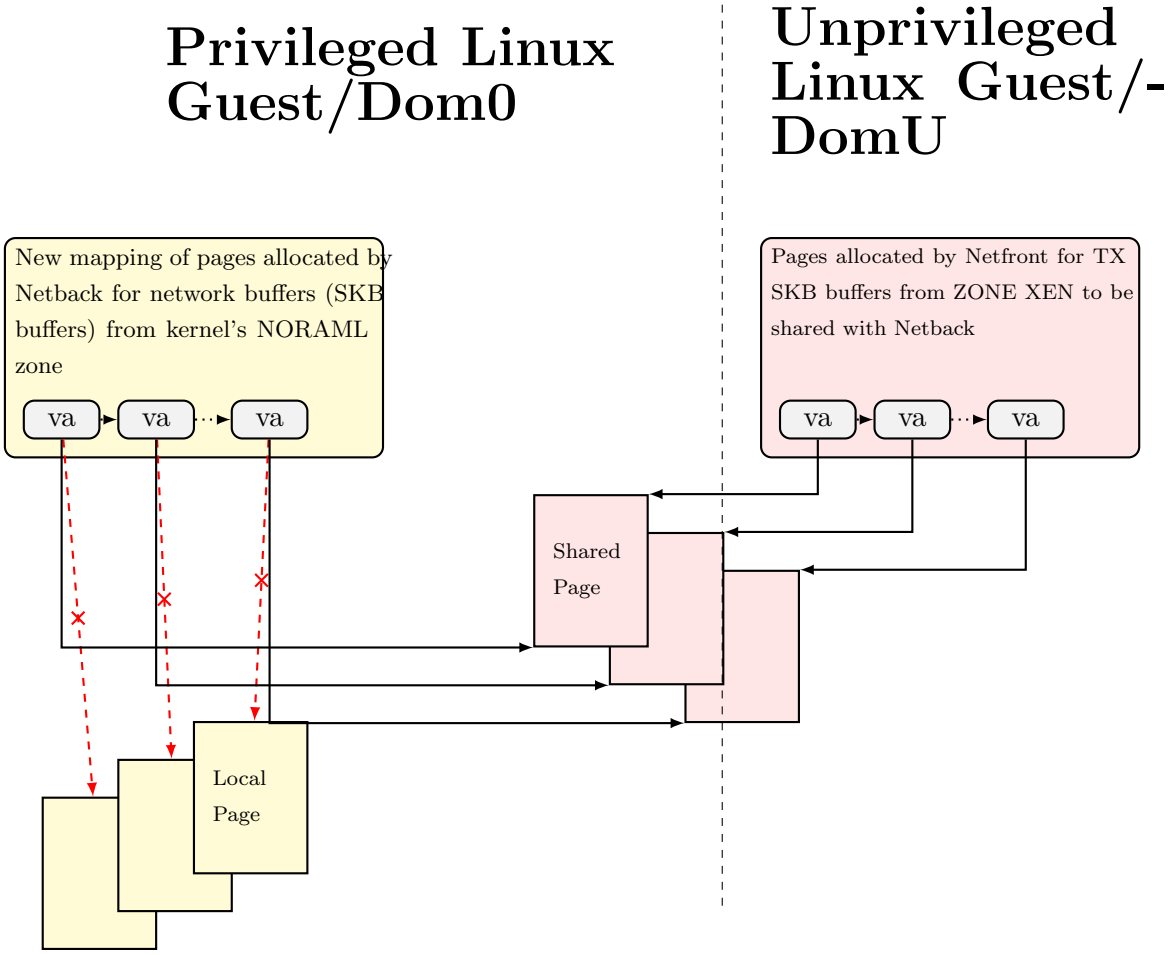

Figure 8.1.: Approach for remapping local virtual addresses by Netback to point to pages shared by Netfront on PHIDIAS

Deployment in high end embedded systems for comparing Xen and PHIDIAS
Limiation of the study

List of Acronyms

I/O	Input and Output
KVM	Kernel-based Virtual Machine
OSes	Operating Systems
PHIDIAS	Provable Hypervisor with Integrated Deployment Information and Allocated Structures
PV	Paravirtualization
HVM	Hardware-assisted Virtualization
Dom0	Domain-0
DomU	Domain-U
ELVIS	Efficient and scaLable paraVirtual I/O System
CDNA	Concurrent Direct Network Access
SR-IOV	Single Root I/O Virtualization
CTSS	Compatible Time Sharing System
Project MAC	Multiple Access Computer, Machine Aided Cognitions
HVAC	Heating, ventilation, and Air Conditioning
VMM	Virtual Machine Monitor
RHEV	Red Hat Enterprise Virtualization
IBM	International Business Machines
EC2	Amazon's Elastic Compute Cloud
OS	Operating System
RTOS	Real Time Operating System
ARM	Acorn RISC Machine or Advanced RISC Machine
RISC	Reduced Instruction Set Computing
MMU	Memory Management Unit
TLB	Translation Lookaside Buffer
KB	Kilo Bytes
IPA	intermediate physical address
GT	Generic Timer
GIC	Generic Interrupt Controller
SMMU	System Memory Management Unit
VM	Virtual Machine
VTLB	Virtual Translation Lookaside Buffer
Schism	Static Configurator for Hypervisor-Integrated Scenario Metadata
QEMU	Quick EMUlator
PVHVM	PV-on-HVM drivers
PVH	PV with Hardware virtualization
XenGT	intel Graphic card
IRQ	Interrupt ReQuest

IPI	Inter-Processor Interrupt
ABI	Application Binary Interface
TDB	tree database
SGI	Software Generated Interrupt
NIC	Network Interface Controller
NAT	Network Address Translation
TX	Transmission
RX	Reception
IPC	Inter-Processor Communication
MTU	Maximum Transmission Unit
DTS	Device Tree Structure

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] I. Spiceworks, "Server virtualization and os trends." https://community.spiceworks.com/networking/articles/2462-server-virtualization-and-os-trends.

[12] "Virtio," Oct 2013. https://wiki.libvirt.org/page/Virtio.

[13] A. Gordon, N. Harel, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards exitless and efficient paravirtual i/o," *Proceedings of the 5th Annual International Systems and Storage Conference on - SYSTOR 12*, 2012.

[14] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system.," in *USENIX Annual Technical Conference*, vol. 26, pp. 231–242, 2013.

[15] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel, and P. Willmann, "Concurrent direct network access for virtual machine monitors," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[16] T. Shinagawa, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, K. Kato, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, and et al., "Bitvisor," *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE 09*, 2009.

[17] H. Raj and K. Schwan, "High performance and scalable i/o virtualization via self-virtualized devices," in *Proceedings of the 16th international symposium on High performance distributed computing*, pp. 179–188, ACM, 2007.

[18] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[20] B. D. Payne and W. Lee, "Secure and flexible monitoring of virtual machines," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007.

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

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

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

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

[25] 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.

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

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

[28] 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.

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

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

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

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

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

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

[35] "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.

[36] "Changing between aarch64 and aarch32 states." http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dui0801a%2FBABBDFIH.html.

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

[38] 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.

[39] 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.

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

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

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

[43] 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.

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

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

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

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

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

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

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

[51] 96boards, "96boards/linux," Jan 2016. `https://github.com/96boards/linux/tree/android-hikey-linaro-4.1`.

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

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

[54] 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`.

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

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

[57] "Chapter 2 describing physical memory."

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

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

[60] "Ip fragmentation," June 2017. https://en.wikipedia.org/wiki/IP_fragmentation.

[61] "Iperf," Aug 2017. https://en.wikipedia.org/wiki/Iperf.

[62] rjmcmahon, "iperf2." https://sourceforge.net/projects/iperf2/files/.

Appendix A

A. Patch for Linux 4.11_rc2

Patch for changes in Linux kernel 4.11_rc2 used for current work is placed at `https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/patch_linux_4.11_rc2`.

Config file used for building Linux executable image for Dom0 guest is placed at `https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/config_dom0_usb`.

Config file used for building Linux executable image for DomU guest is placed at `https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/config_dom1_usb`.

Command used to cross compile Linux image is given below:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image
```

Listing 1: Command to cross compile Linux image for ARMv8 target on x86_64 host

B. DTS file for HiKey ARMv8 Board

DTS file for HiKey ARMv8 target platform used is placed at `https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/hi6220new1.dts`.

Command used to compile dts file is given below:

```
dtc -I dts -O dtb -o hi6220new1.dtb hi6220new1.dts
```

Listing 2: Command to compile dts file into dtb

C. Patch for PHIDIAS changes

Patch for changes in PHIDIAS, related to adding serial multiplexer support for two guests running on different physical cores (0 and 1), for current work is placed at

https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/patch_phidias.

Scenario file used for building image for two guests (Linux1 for Dom0 and Linux2 for DomU) on PHIDIAS is placed at https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/scenario.xml.

D. Patch for Xenstore Tool changes

Patch for changes in Xenstore are placed at https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/patch_xen.

E. initramfs source code

Initramfs image was used by PHIDIAS's configuration framework for building final executable image in our current work. The source of initramfs is placed at https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/tree/master/initrd.

For every modification in initramfs files, we have to recreate image in gzip format.The changed size of resultant initramfs image should be then placed in hi6220new1.dts file in chosen node as shown in listing 3.

```
 chosen {
 stdout-path = "serial3:115200n8";
 bootargs = "console=ttyAMA3 earlycon=pl011,mmio32,f7113000 root=/dev/ram0
     cmdline_from_dt init=/bin/sh loglevel=8";
 linux,initrd-start = <0x0 0xa000000>;
 linux,initrd-end = <0x0 0xAEA9B59>;
 };
}
```

Listing 3: Chosen node in dts file for specifying initial ramdisk size

For this, a script has been developed which recreates the initramfs image, pack all files into an final initramfs file in gzip format and calculates the size which should be specified in **linux,initrd-end** option of chosen node in used dts file. This script is placed at https://gitlab.sec.t-labs.tu-berlin.de/a.waseem/Thesis/blob/master/script_size.sh.

Appendix B : Running Xenstore and Enabling Virtual Network Interfaces on PHIDIAS

After loading final image, built by PHIDIAS configuration framework, on HiKey ARMv8 target, following commands in Dom0 guest are used to mount Xen Filesystem and start Xenstore daemon:

```
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 4: Commands to mount xenfs and start Xenstore daemon

The next step is to probe netback driver in Dom0 guest. For this, a script **script_xenstore.sh** has been written and placed in initramfs file. It is run using the following command:

```
#sh script_xenstore.sh
```

Listing 5: Running script to probe netback driver in Dom0

After successful running of the script, vif1.0 interface will be created in Dom0. For triggering the probe of netfront driver in DomU, we have to send an IPI to DomU guest. For this, a kernel driver is developed which is also placed in initramfs file under directory 'trugger'. Following command inserts the module in Dom0 kernel which triggers the IPI to DomU:

```
trigger# insmod testinit.ko
```

Listing 6: Running driver to probe netfront in DomU

After successful completion of above command, a virtual network device named eth0 will be created in DomU. Before using these virtual interfaces for communication between Dom0 and DomU, following commands are issued to enable the devices and set their addresses:

```
On Dom0:
#ifconfig vif1.0 hw ether 82:80:48:BA:d1:30
#ifconfig vif1.0 10.0.0.1 netmask 255.255.255.0
```

```
#ip link set dev vif1.0 up

On DomU:
#ifconfig eth0 10.0.0.2 netmask 255.255.255.0
#ip link set dev eth0
```

Listing 7: Enabling and setting addresses of virtual network interfaces