BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

VIRTUALIZATION OF I/O OPERATIONS IN COMPUTER NETWORKS

VIRTUALIZACE VSTUPNÍCH A VÝSTUPNÍCH OPERACÍ V POČÍTAČOVÝCH SÍTÍCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN REMEŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. DENIS MATOUŠEK**
VEDOUCÍ PRÁCE

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Remeš Jan, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Virtualizace vstupních a výstupních operací v počítačových sítích**
Virtualization of I/O Operations in Computer Networks
Kategorie: Počítačové sítě

Pokyny:

1. Seznamte se s technologiemi používanými pro virtualizaci vstupních a výstupních operací v počítačových sítích.
2. Zaměřte se na technologii SR-IOV a nastudujte související prvky standardu PCI Express.
3. Navrhněte implementaci podpory technologie SR-IOV pro zvolenou referenční platformu.
4. Na zvolené platformě proveďte referenční implementaci technologie SR-IOV.
5. Proveďte testy referenční implementace na zvolené platformě ve virtuálním prostředí, zhodnoťte výsledky a diskutujte možnosti dalšího rozšíření.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matoušek Denis, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstract

This work deals with virtualization of computer systems and network cards in high-speed computer networks, and describes implementation of the SR-IOV virtualization technology support in the COMBO network card platform. Various approaches towards network card virtualization are compared, and the benefits of the SR-IOV technology for high performance applications are described. The work gives overview of the COMBO platform and describes design and implementation of the SR-IOV technology support for the COMBO platform. The work concludes with measurement and analysis of the implemented technology performance in virtual machines. The result of this work is the COMBO cards' support for the SR-IOV technology, which makes it possible to use them in virtual machines with wire-speed performance preserved. This allows future COMBO cards to be used as accelerators in the networks utilizing the Network Function Virtualization.

Abstrakt

Tato práce se zabývá problematikou virtualizace počítačových systémů a zejména síťových karet ve vysokorychlostních sítích, a řeší implementaci podpory virtualizační technologie SR-IOV pro síťové karty COMBO. V práci jsou shrnuty různé přístupy k virtualizaci síťových karet a popsány výhody technologie SR-IOV pro vysoce výkonné aplikace. Dále práce obsahuje informace o platformě COMBO a popisuje návrh a implementaci podpory technologie SR-IOV pro tuto platformu. Závěrem je provedeno vyhodnocení výkonnostních testů implementované technologie ve virtuálních strojích. Výsledkem práce je podpora technologie SR-IOV v kartách COMBO, což umožňuje jejich použití ve virtuálních strojích při zachování vysokého výkonu. To umožní budoucím COMBO kartám fungovat jako akcelerátory v sítích využívajících virtualizace síťových funkcí.

Keywords

Computer networks, network card, NIC, virtualization, virtualization technologies, SR-IOV, COMBO, NetCope, firmware, drivers, FPGA, NFV

Klíčová slova

Počítačové sítě, síťové karty, virtualizace, virtualizační technologie, SR-IOV, COMBO, NetCope, firmware, ovladače, FPGA, NFV

Reference

REMEŠ, Jan. *Virtualization of I/O Operations in Computer Networks.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Denis Matoušek

Virtualization of I/O Operations in Computer Networks

Declaration

Hereby I declare that this master thesis was created as an original author's work under the supervision of Mr. Denis Matoušek. Supplementary information was provided by Mr. Martin Špinler. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jan Remeš
May 24, 2017

Acknowledgements

First of all, I would like to thank my supervisor, Mr. Denis Matoušek, for supervising my thesis and providing me with excellent advice and support.

I would also like to thank Mr. Martin Špinler, who has spent large amounts of time providing me with technical support and making SR-IOV firmware support possible, and all the people at CESNET for allowing me to work with them and use their hardware.

Finally, I must thank Mr. Petr Kaštovský, Mr. Tomáš Závodník and others from Netcope Technologies for being helpful and supportive, and providing me with great working environment.

Contents

Chapter 1

Introduction

This work deals with virtualization of network input/output (I/O) devices in computer systems – it describes I/O virtualization and common approaches to it, and the SR-IOV technology. It continues with basic description of the COMBO network card platform and describes implementation of the SR-IOV technology support for the COMBO cards.

I/O virtualization is usually a part of more general concept – *hardware virtualization*. Hardware virtualization (or just *virtualization*) is understood as creating entire virtual computers (machines) and running applications – and even entire operating systems – in the virtual environment. These *virtual machines* do not have access to physical hardware. For a virtual machine to perform network input/output operation, a virtual network input/output device has to be created – I/O virtualization is needed.

Existing technologies for I/O virtualization differ in performance, flexibility and ease of use. They are therefore suitable for different use cases. The SR-IOV technology provides high performance at the cost of reduced flexibility, it is therefore suited for performance-critical usage scenarios.

Chapter 2 describes the details, benefits and general motivation for using virtualization, and I/O virtualization in particular, as well as describe its future application in the form of *Network Functions Virtualization* (NFV).

Chapter 3 describes how input and output network traffic is processed by individual components in the computer system, in order to allow for comparison of individual virtualization technologies.

Chapter 4 starts on theoretical background, describing the concept of *virtual memory*. Then it proceeds to enumerate existing approaches towards I/O virtualization, giving examples of technologies utilizing said approaches. It is concluded with challenges in the I/O virtualization and the overview of the individual approaches and their advantages.

Chapter 5 provides an in-depth description of the *SR-IOV* I/O virtualization technology, including the description of its requirements for the NIC firmware and its driver.

In chapter 6, the COMBO platform — developed as a part of the Liberouter project[1] — is described, with focus on its components relevant for the SR-IOV technology.

Chapter 7 describes how the SR-IOV support for the COMBO cards was designed and implemented. Performance evaluation is conducted in chapter 8.

Installation steps are described in appendix A.

[1] http://www.liberouter.org/technologies/cards

Chapter 2

Motivation

2.1 Virtualization

Virtualization is a technology for creating and running (multiple) *virtual machines* (virtual computers, also referred to as *guests*) on a single physical machine (also referred to as a *host*). The host runs a virtualization manager program (a *hypervisor*) which creates and manages individual virtual machines. The virtual machines, isolated from each other, run their own operating systems and software. Closer description of virtualization is given by [14], detailed information and hypervisor comparison can be found in [9].

Many organizations run multiple network services (HTTP server, DNS server, etc.) both for their internal use and for the public. For security reasons, service independence and individual service requirements, it is widely adopted best-practice to run each of these services on a separate machine (server).

In non-virtualized environments, this requirement results in a large number of physical devices being:

- powered
- connected to physical network
- located in a server room
- managed

If high availability is required for a given service, the administrator must buy, install and manage yet another machine, potentially doubling the machine count.

In non-virtualized environment, servers will likely waste a large percentage of their computational power (CPU time), as their load will be low for most of the time. Each server must be powerful enough to handle its service's peak load, wasting this performance when the load is lower (likely most of the time). Furthermore, performance required for many services (such as DNS server in a small company) is much lower than performance offered even by entry-level servers, leading to even bigger inefficiency.

If a physical machine stops responding (due to network misconfiguration, network interface going down, or system error), the administrator needs to reboot it. This can be done manually, or via remote access technologies, like Intel's IPMI[1] or Dell's iDRAC[2].

[1] See https://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface

[2] See https://www.dell.com/learn/us/en/555/solutions/integrated-dell-remote-access-controller-idrac

The technologies allow for better manageability, yet they have to be separately configured, maintained and provide a possible security hole[3].

In virtualized environments, however, it is possible to run each service on an independent *virtual machine*, and many such virtual machines can be hosted on a single physical server. This resolves issues mentioned above:

- Single rack space is required
- Single physical network connection is required
- Single server will have greatly reduced power requirements
- Resource utilization is better, as computational power is shared among the services, and since their load peaks are unlikely to coincide, the server need not be powerful enough to handle all its services' peak loads simultaneously
- Management is easier, since starting, stopping, resetting, cabling etc. can be done in the hypervisor and does not require physical access to the servers nor specialized interfaces like IPMI or iDRAC
- Virtual machines can be snapshotted, backed up, restored, or migrated to another physical server for much less cost (in time and resources spent to the task)

High availability must still be ensured by duplicating the physical server, but duplicating one server is far preferable to duplicating dozens of service servers.

To conclude, virtualization reduces the cost of deployment, provides better flexibility and eases server management. For these reasons, it is now widely used.

2.2 I/O virtualization

All computer systems are usually required to perform some kind of I/O operations. For this, I/O resources (I/O capable hardware) are required.

In a virtualized environment, physical resources (CPUs, RAM, hard drives, network cards, etc.) are shared among the virtual machines. Since the virtual machines must not affect each other's environment, they must be in fact isolated from the real hardware, or share it on the basis of exclusive ownership (that is, each resource can be assigned to — and used by — a single machine only).

CPU cores can be shared in the same manner they are shared between running processes in multiprocessing system – each virtual machine can be assigned the CPU for a given time slice. RAM can be shared exclusively – each virtual machine may be given a portion of the physical RAM and the Memory Management Unit[4] (*MMU*) will provide mapping from each machine's virtual address space to physical address. This is discussed in detail in section 4.1.

Sharing an I/O hardware is an issue, since such hardware is usually designed for being used by a single system. In many cases, sharing it is just impossible – if a hard drive was shared, individual virtual machine's writing operations would affect the state of other virtual machines, which is unacceptable.

Network cards sharing is problematic for following reasons:

[3] See http://www.itworld.com/article/2708437/security/ipmi--the-most-dangerous-protocol-you-ve-never-heard-of.html

[4] https://en.wikipedia.org/wiki/Memory_management_unit

- They contain host-specific settings, such as physical address used for incoming packet filtering
- They must actively deliver inbound packet to the system memory – a switching logic would have to select proper addressee

For exclusive ownership sharing, N physical network cards (where N is the count of virtual machines the server ought to run) must be installed on the server. With modern servers hosting dozens or even hundreds of virtual machines (for example, ESXi vSphere 6.5 allows maximum 1024 machines[5]), this option is very inefficient.

As a result, I/O virtualization must be performed in virtualized environments.

2.3 Network Function Virtualization

Network Function Virtualization (*NFV*) is a proposal of network architecture published by ETSI [2]. The main idea in the proposal is that network functions (such as firewall, load-balancing, tunneling, etc.) should run in software on dedicated virtual machines using commodity hardware.

Today, these functions are usually provided by specialized hardware boxes, especially in large networks. These dedicated devices are usually very expensive[3] due to the fact their market is much smaller than that of commodity hardware.

Running a network function as a virtual machine provides significant benefits to the network administrators, namely:

- Reduced cost (cheaper hardware, less maintenance)
- Flexibility (installation, modification, upgrade, relocation)

NFV has a major requirement for the virtual machines, though, and that is *performance*. Multiple virtual network functions running on a single server operating on 10 Gb/s (or more) network will collectively have to process tens of millions of packets per second.

In such an environment, the requirements for I/O virtualization are:

- to provide close to bare-metal performance, and
- to avoid performance bottle-necks (e.g. extreme load on the hypervisor CPU)

[5] https://www.vmware.com/pdf/vsphere6/r65/vsphere-65-configuration-maximums.pdf

Chapter 3

I/O processing

This chapter describes the process of receiving and transmitting packets for network applications running in Linux. Understanding the process is important for explanation of the differences between I/O virtualization technologies.

For the purpose of this work, network applications are divided in two groups: *Regular* and *High Performance* (HiPerf) applications.

A *regular* application is one that utilizes system network I/O capabilities for communication with a remote site. Most of the classical network applications (web browsers, e-mail clients, on-line video games, etc.) fall in this category, as well as their corresponding servers. These applications usually open a TCP or UDP socket, and send *(and receive)* data through it.

A *high performance* application works on the link layer of the ISO/OSI model. It uses system's networking services to receive *(and send)* individual frames. Applications like firewalls, network monitoring tools, or packet analyzers fall in this category. These applications usually open a *raw socket* (SOCK_RAW) or use specialized frame capturing mechanisms like DPDK (covered in section 3.5). The applications require higher performance, since they analyze entire network traffic, and often run on a dedicated machine in order to be able to fully utilize its resources.

3.1 Receiving data (RX direction)

When a frame is received by the NIC (network interface card), it fires an interrupt (*RX interrupt* in figure 3.1) to notify the system about it. Subsequently, the kernel calls corresponding driver's polling function to receive the data. This is usually achieved through DMA transfer from the NIC buffer to a kernel buffer (see figure 3.2).

Inside the kernel, the frame is parsed, its L2 to L4 headers analyzed and acted upon (ICMP response, TCP acknowledgment, etc.) if necessary. The destination application (or rather, destination socket) is chosen here, if one exists.

Finally, the payload is extracted and delivered to the application's socket. If the application has issued *recv()* call before, it has been blocked, and gets waked at this moment. Otherwise, the data will be returned when the application issues the call.

If the application is HiPerf, its data will consist of the entire frame as seen on the wire (the delivery is done before parsing).

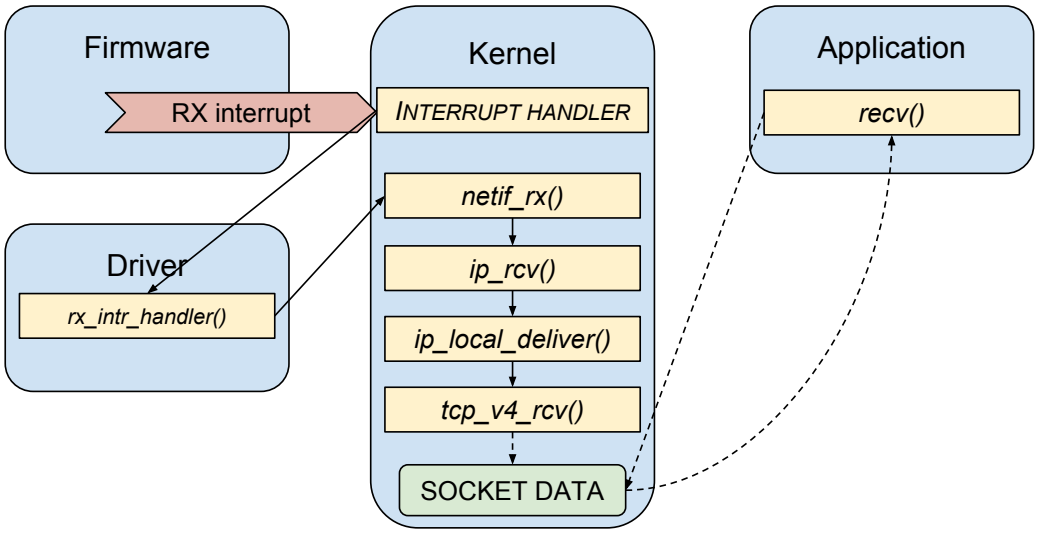


Figure 3.1: RX Packet Processing (Linux kernel network stack function call graph)

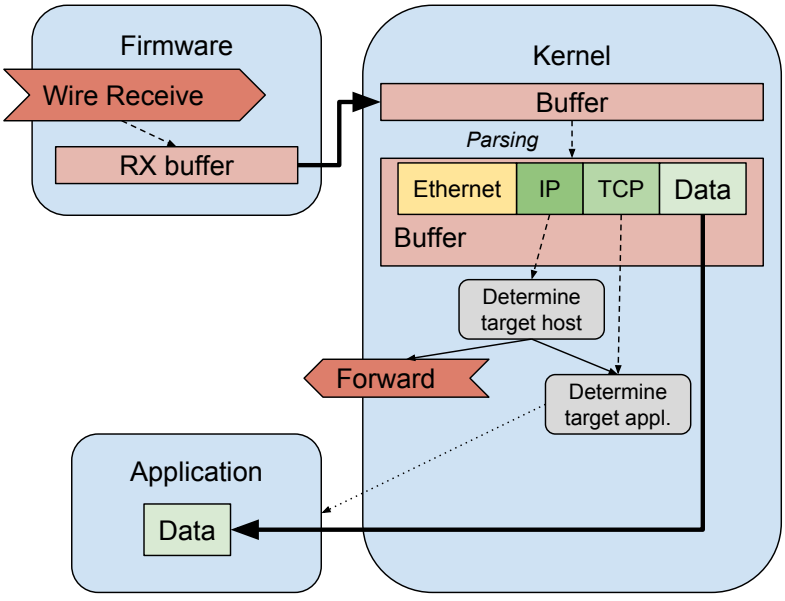


Figure 3.2: RX Packet Processing (data flow)

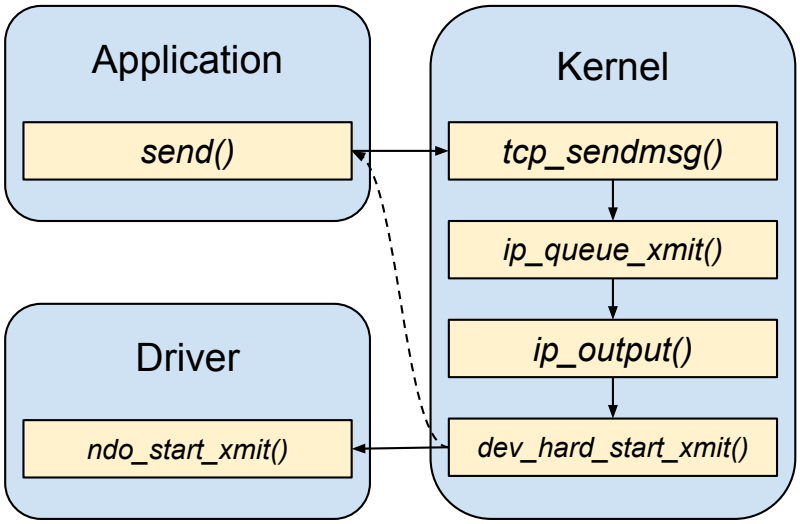


Figure 3.3: TX Packet Processing (Linux kernel network stack function call graph)

DMA

Direct Memory Access is a technology for data transfers from/to the main memory independent of the CPU.

This technology alleviates the CPU of the need to control the transfer and allows it to do other work when the data is copied.

Older architectures required so-called DMA controller to perform DMA operations, but modern PCI devices are able to do so on their own. Therefore, with DMA-capable NIC, its driver can allocate a DMA buffer in the memory and pass its address to the device, and it will copy the data there.

3.2 Transmitting data (TX direction)

Data transmission starts with the application issuing a *send()* call (see figure 3.3 for call graph). Transmitted data is copied to the kernel (see figure 3.4), then L4 to L2 headers are added and the right NIC for transmission is chosen based on routing tables.

When the frame is assembled and placed in a buffer, the driver of the chosen NIC is called to transmit the data. It copies the buffer to the NIC using DMA and instructs the firmware to transmit it.

If the HiPerf application sends data, it skips the protocol stack (adding headers and routing) by using the raw socket.

3.3 Driver processing

The NIC driver registers its associated device as a networking device, providing among others the *receive* and *transmit* method.

When there is data to be transmitted, the driver's transmit function is called by the kernel. The driver's code ensures the data is copied to the NIC via DMA.

On reception, kernel (after being notified by interrupt) will call driver's receive method to copy the inbound packet's data from the NIC to the kernel buffer (see figure 3.1).

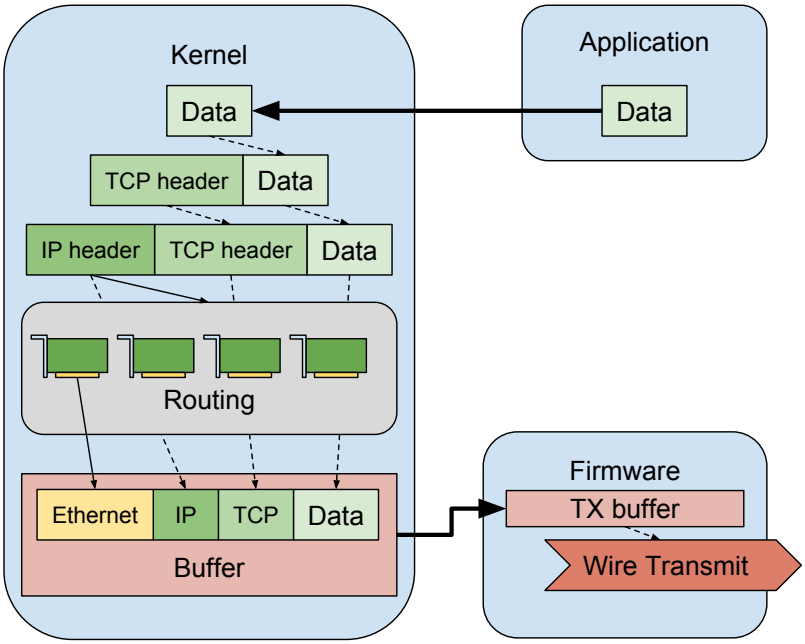


Figure 3.4: TX Packet Processing (data flow)

3.4 NIC processing

The firmware in the NIC processes packets provided by the system (e.g. calculates their checksums) and transmits them on the physical medium.

At the same time, when it receives a packet from the medium, it stores it in a local memory and fires an interrupt to notify the system of the new packet.

3.5 DPDK

Data Plane Development Kit is a network framework for fast packet processing developed by Intel. High Performance Applications using DPDK can achieve higher throughput due to DPDK's features described below. It should be noted that DPDK is unsuitable for *Regular Applications*, since it does not provide the abstraction (the application has to work with frames) and, being a framework, it requires the application to be written specifically for the DPDK backend.

3.5.1 Kernel bypass

DPDK applications run completely in user-space, therefore avoiding the need for costly context switches between the application and the kernel. To achieve that, DPDK employs (see figure 3.5):

- user-space polling-mode NIC drivers (PMDs)
- internal memory management utilizing **hugepages**

thus avoiding the need to call kernel during the application run.

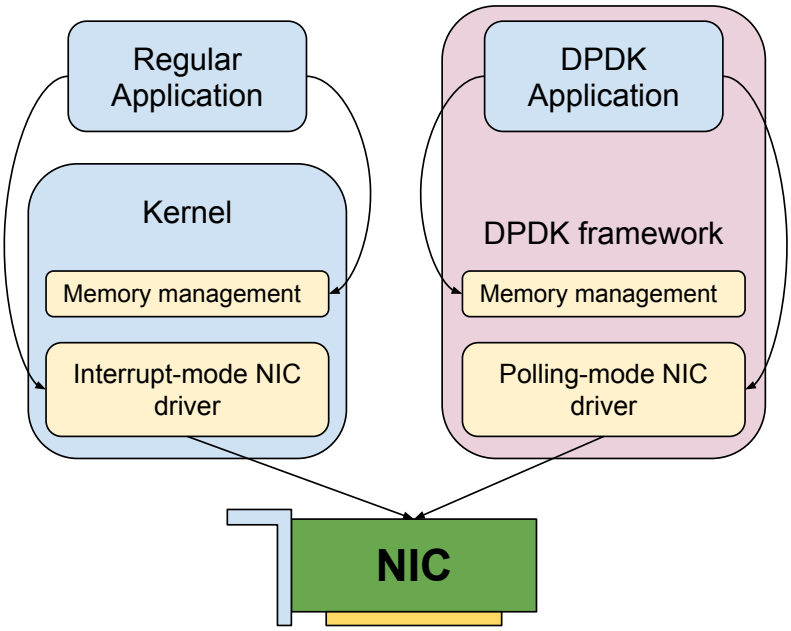


Figure 3.5: DPDK Kernel Bypass

3.5.2 Polling Mode drivers

Interrupt driven I/O is prevalent in today's computer platforms, because the interrupt mode has removed the need for periodical checking of I/O devices' state, leaving the CPU to do useful work when there are no I/O events.

However, in an environment where the platform's single task is incoming packet processing on Gb/s link speeds and high performance is required, sheer amount of interrupts per second and the overhead necessary for handling them become significant.

Polling-mode drivers allow the application to check for incoming data in an active spin like this:

```
unsigned char * data = NULL;

do {
data = driver_get_next_data();
} while(data == NULL);
```

Since DPDK works in user-space, it requires the drivers used to be polling-mode, since interrupts would switch the context to the kernel, which is undesirable.

Chapter 4

I/O virtualization

In chapter 3, network I/O processing on a bare-metal (non-virtualized) system with a physical NIC was described. This chapter will elaborate on approaches of virtualizing the NIC and their differences. Rosenblum[10] gives an overview of I/O virtualization benefits and provides the logical view on virtual NICs from the guest and the host view.

The goal of the I/O virtualization technologies is to create a virtual NIC (*vNIC*) in the virtual machine. They aim to achieve high performance (throughput and low latency) of the vNIC and high flexibility (e.g. allowing virtual machine *migration* to another physical host).

In the first section, memory virtualization will be described, since it is essential to the understanding of NIC virtualization technologies. Later, starting with section 4.2, individual technologies will be presented.

4.1 Virtual memory

Physical memory (RAM) installed in a computer system can be conceptually seen as a contiguous array of bytes, indexed from zero to the physical RAM's size. Modern computer systems, running multiple processes need a way to share the RAM among them, which is made difficult by the fact that total memory requirements of each process are not known and that the process may allocate or free memory at any time.

Virtual memory addresses this issue. Each process is provided 4 GB (or more with 64bit systems) of its own contiguous **virtual** memory, indexed from zero. Physical RAM is split into fixed-size (usually 4 KB) *pages*, and a mapping

$$(program, programPage) => physicalPage$$

is established in the operating system (see figure 4.1). When such program's memory is accessed, kernel installs appropriate translation in a hardware circuitry called **MMU** (Memory Management Unit). The memory address used by the application is split – the upper bits serve as an index to the translation table, while the lower bits form an offset within the page. For example, an access to the address `0x000001a0` made by Application A from figure 4.1 would result in access to physical RAM address `0x6ca0411a0`. Upon context switch, the MMU is instructed by the kernel to use mapping for the process switched to, so that every memory access from that program is translated (mapped) by the MMU to use the right page.

Since the virtual machines are seen as processes by the host, they are assigned their virtual memory and MMU mapping like any other. There are MMUs which offer hardware

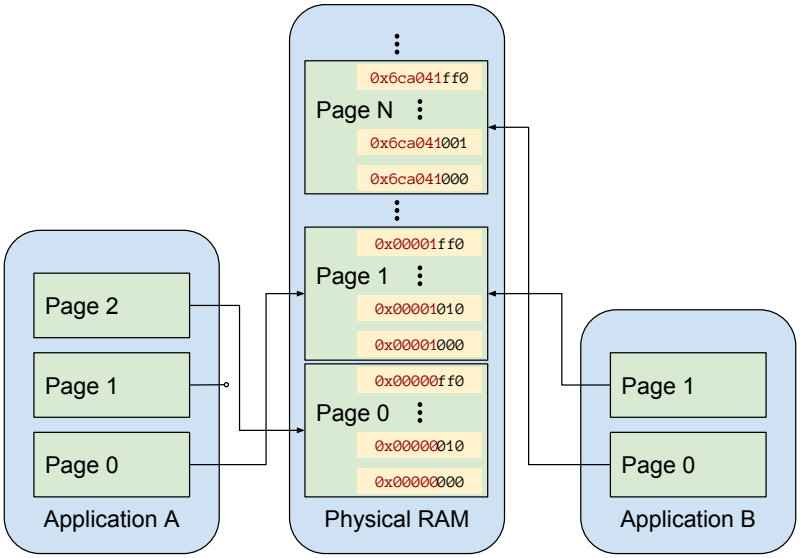


Figure 4.1: Virtual memory mapping and sharing

support for virtualization, and those allow the virtual machine to set its own mappings inside its virtual address space, like the physical machine would do with physical RAM (see figure 4.2). *Where such hardware support is missing, the hypervisor must emulate it.*

During a DMA transfer, the I/O device provides the memory address the data are to be copied to/from. When the device is assigned to a virtual machine, the address (programmed to it by the virtual machine driver) will be the address as seen by the virtual machine. The hypervisor must ensure – using hardware circuitry called **IOMMU** (Input/Output MMU) – that the memory accesses from a device assigned to a VM undergo that VM's MMU translation.

4.2 NIC emulation

Emulation is the slowest, but most reliable technology of NIC virtualization. The hypervisor vendor chooses a well-known hardware product (such as *AMDPCNet Fast III*[1]), supported by most of the major operating systems, and emulates its behavior in software. The advantage of this approach is compatibility with most of the operating systems, but the overhead required may cause severe performance drops.

Software emulation can be hardware-accelerated using technologies like VMDq (*Virtual Machine Device Queues*), which makes the physical NIC classify inbound packets into queues for individual virtual machines, removing the need to steer them during software processing. This improves throughput, scalability and capacity of the system[13]. VMDq technology overview[12] provides further information.

4.3 Paravirtualization

Paravirtualization refers to virtualization where the virtual system is to a certain level aware that it is virtualized. Paravirtualization of I/O devices is similar to NIC emulation described

[1]See https://www.virtualbox.org/manual/ch06.html

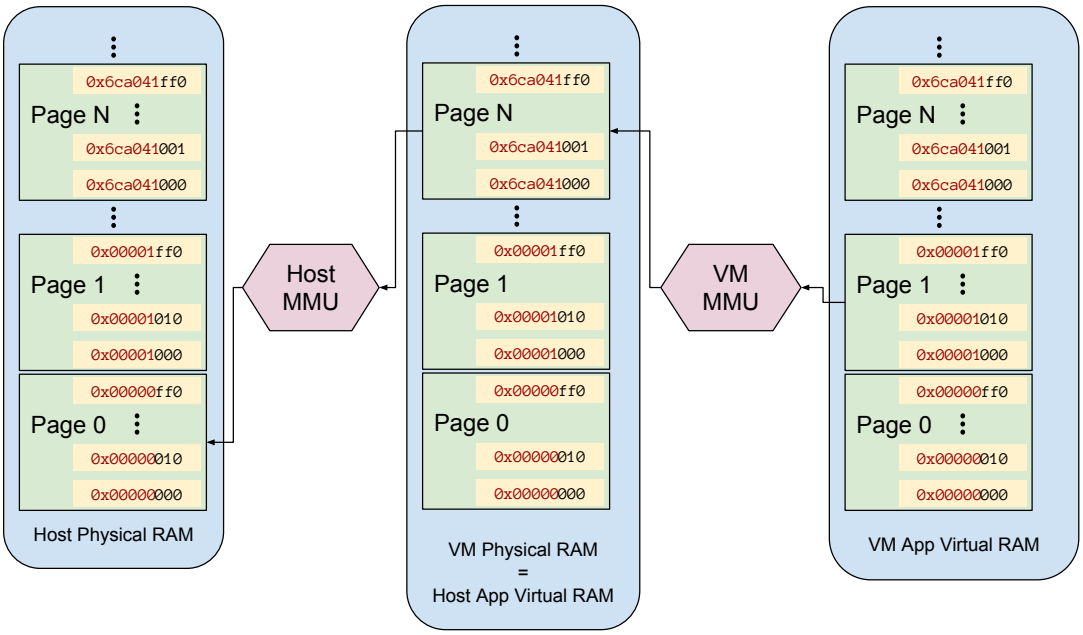


Figure 4.2: Multi-level virtual memory mapping

in section 4.2, but instead of well-known hardware model, a special *virtual network card* is emulated.

Paravirtualized drivers consist of *frontend driver* running in the guest, driving the virtual device; and *backend driver* running in the host (usually as part of the hypervisor), providing the virtual device's services to the guest. Figure 4.3 shows the drivers in the overall schema.

Since the NIC is designed to be emulated, it can offer different functionality and API than regular NICs. It is therefore designed with virtualization efficiency in mind, potentially yielding superior performance to the emulated NICs.

4.3.1 virtio

Virtio is common layer for para-virtualized devices in Linux. It was created by Rusty Russell with the aim to simplify writing virtual device drivers and to reduce the amount of code present in the drivers.

Russell [11] describes virtio as a three-part platform: it provides frontend drivers, transport mechanisms (virtqueues) and configuration access. These can be understood as an API, and hypervisor vendors can implement backends for their devices using this API.

Jones [5] shows how virtio architecture leverages the performance, reducing number of per-packet traps.

4.4 NIC virtualization

Both emulation and paravirtualization create the *vNIC* in software. With (full) NIC virtualization, an actual hardware device (or hardware-emulated virtual hardware device) is assigned to the virtual machine, so that it communicates with it directly, without the hypervisor or the host operating system being involved (see figure 4.3).

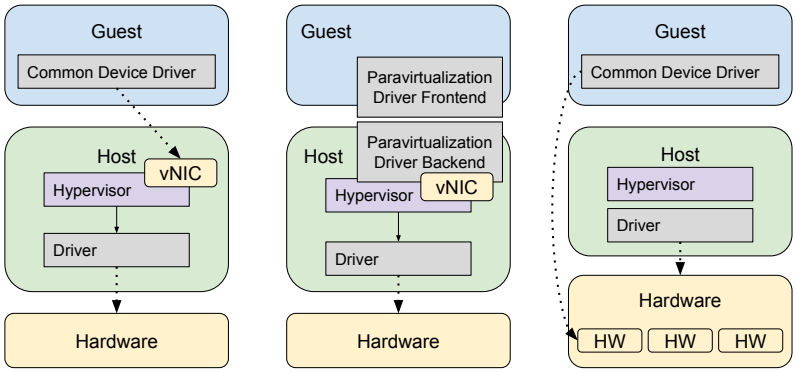


Figure 4.3: Comparison of virtualization technologies

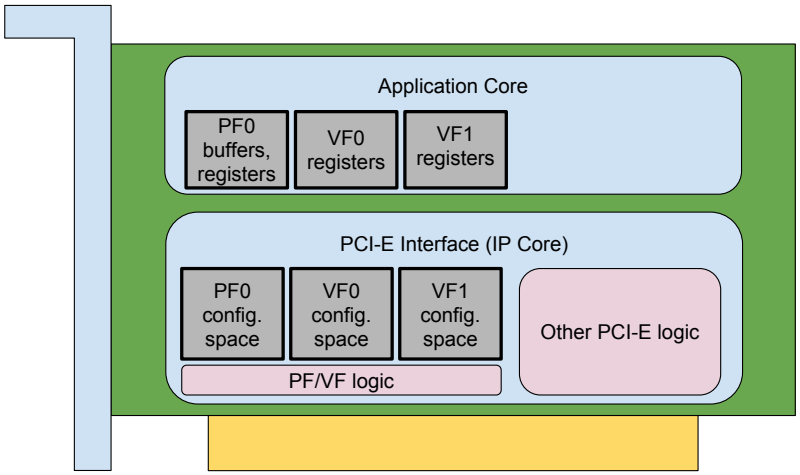


Figure 4.4: SR-IOV hardware resources

This approach yields the best performance, since the virtual machine communicates directly with the hardware, but there are caveats – either the physical machine has to have enough physical NICs for all the virtual machines, or hardware-level NIC virtualization must be performed. Industrial standard for hardware-level virtualization is the technology called SR-IOV.

4.4.1 SR-IOV

SR-IOV is a full NIC virtualization technology developed as a PCI standard by PCI SIG (Special Interest Group) consisting of major hardware vendors. It allows a PCI device which supports it to present itself to the system as multiple devices on the PCI bus level. Although such devices (called *Virtual Functions* as opposed to the original device called *Physical Function*) have limited capabilities, they may serve as NICs and may be assigned to individual virtual machines.

The device must contain hardware resources (buffers, descriptors, etc.) for each virtual function. Therefore, a device can only support so many virtual functions (usually 8, 64 or 256). Figure 4.4 shows example card firmware with resources for a single physical and two virtual functions. Furthermore, its firmware must be able to process requests from various machines simultaneously. If used for NIC virtualization, the device also needs to implement

some kind of switch to determine which virtual machine should receive the packets delivered on the physical medium.

Technology primer[7] from Intel gives first insight into SR-IOV, the technology is closer described in chapter 5.

4.5 Virtualization challenges

I/O virtualization faces challenges associated mainly with two changes it introduces:

1. Extending the network by adding multiple new network interfaces (vNICs)
2. Binding virtual machine with the I/O hardware

Extending the network requires some form of packet switching, while binding hardware with the virtual machines poses a challenge for virtual machine migration – their transfer to another physical host.

4.5.1 Switching

For a computer system running on bare-metal with a single interface NIC, network topology and behavior are largely external issues. The system's NIC forwards to it packets matching its physical address or broadcast packets, and drops all others.

When an I/O virtualization technology creates multiple network interfaces, the hypervisor or the host system must connect the vNICs to the network in some manner. The requirements can differ ("seamless" bridged connection to the outer network, host-only network, ...).

In any case, the hypervisor must implement a switching mechanism to decide which packets will be delivered to which virtual machine. One of the possible approaches is to create a virtual switch (e.g. Open vSwitch[2]) and connect the vNICs to it. Switching mechanism is closely bound to the migration problem discussed in the following section.

4.5.2 Migration

Virtual machine migration is the process of transferring a virtual machine between physical hosts. *Offline* migration is performed when the virtual machine is not running, and is relatively straightforward. *Online* migration, on the other hand, is performed while the machine is running and places serious demands on I/O virtualization system (e.g. the requirement to preserve open TCP connections).

Virtualization technologies bind the virtual machine with *vNICs*, which may be difficult to duplicate on the new physical machine. For emulation and paravirtualization (see sections 4.2 and 4.3, respectively), the *vNIC* is software-based and can be recreated on the target machine (provided the hypervisor is the same or at least provides the same functionality). With NIC virtualization, this is not possible, because from the host's perspective, the virtual machine is assigned one of the available physical devices, and this cannot be propagated with the machine – the target machine may well not even have such a device available.

[2] http://openvswitch.org/

4.6 Summary of virtualization technologies

Virtualization technologies described in this chapter use different methods to provide virtual NIC for virtual machines. They achieve different results both in terms of performance and flexibility. This is illustrated in table 4.1.

Emulation is the basic, slowest technology. It is widely supported by the hypervisors, since it works with any guest system which supports the emulated hardware, and does not reveal to the guest that it is virtualized.

Paravirtualization improves performance (while keeping flexibility) by emulating a device designed for virtualization. It requires this device's driver to be present in the guest. Modern hypervisors usually utilize *virtio* technology, and modern systems usually include a driver for the *virtio-net* device, making paravirtualization relatively easy to run.

Full virtualization is designed for performance-critical environments. It requires hardware support and limits flexibility (as the virtual machines are bound with physical hardware), but it offers close to bare-metal performance.

The following chapters deal with SR-IOV, which is the standardized technology of full virtualization for PCI-E devices.

	Emulation	**Paravirtualization**	**Full virtualization**
Performance	Low	Medium	High
Host CPU load	High	Low	None
Complexity	Hypervisor	Hypervisor, Guest	Hardware, Host, Guest
Flexibility	High	High	Low
Technology	Hypervisor-specific	virtio	SR-IOV
NIC example	Intel 82540EM	virtio-net	Mellanox Connect-X 4

Table 4.1: Virtualization technology comparison

Chapter 5

SR-IOV

PCI *Single Root Input/Output Virtualization* is a standard[1] for PCI devices virtualization and sharing. Compliant devices are able to spawn virtual PCI devices, with which the computer system then communicates.

This chapter describes how SR-IOV works, and follows with design requirements for SR-IOV capable device driver and firmware. More information can be found at [8]

5.1 SR-IOV principles

SR-IOV is a PCI-level technology. It allows a physical device to be "seen" as multiple devices on the PCI bus itself. Therefore, it is necessary to understand how PCI devices present themselves on the bus and how they are accessed.

The PCI bus is organized in a hierarchical manner (see figure 5.1). The computer system may contain several *buses*, which may connect several *devices*, and each device may host several *functions*. This distinction is based on understanding that a single physical expansion card may contain multiple individual devices (e.g. USB controller and FireWire controller). Basics of the PCI bus are described in [4, chapter 12].

As a result, the PCI devices use *bus:device.function* addressing, where

[1] https://pcisig.com/specifications/iov?field_document_type_value%5B%5D=specification&speclib=

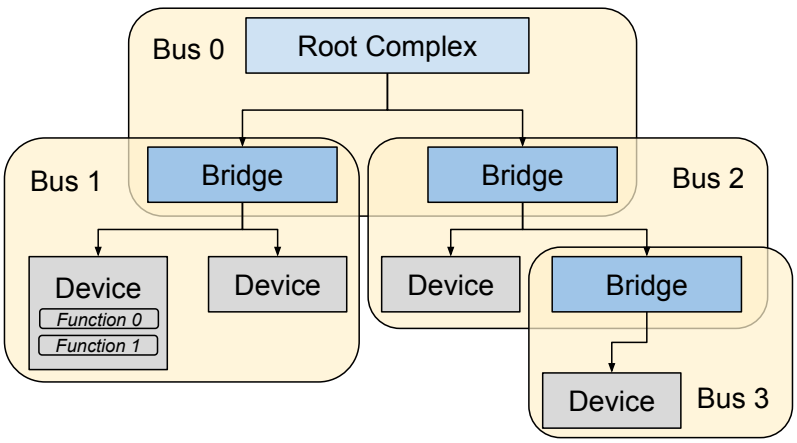


Figure 5.1: Example PCI-E Hierarchy

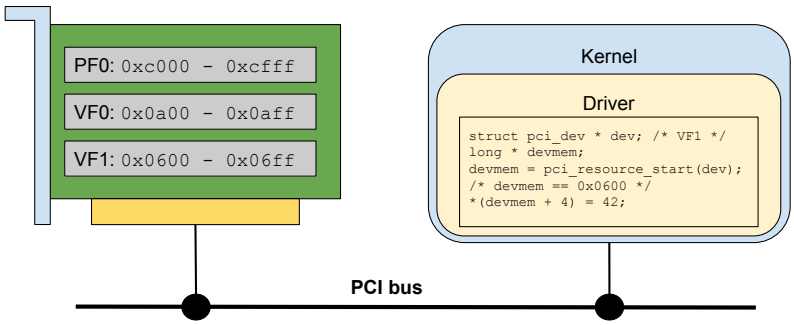


Figure 5.2: SR-IOV memory mapping

- *bus* is a number from *0x00* to *0xff* (0 - 255)
- *device* is a number from *0x00* to *0x3f* (0 - 31)
- *function* is a number from *0x0* to *0x7* (0 - 7)

Example PCI device address would be *01:1a.0*.

The PCI devices have 3 address spaces – configuration, memory an I/O. When the system boots, the BIOS enumerates the devices and maps them to the processor's address spaces. This mapping is written to the device's BAR (*Base Address Register*) registers, so that it knows which address ranges are assigned to it and may respond appropriately.

PCI devices' capabilities are advertised to the system through the list of so-called *capability structures* located in the device's configuration space. These capabilities form a logical linked list, and the system enumerates the list to determine device's capabilities. An SR-IOV capable device contains a SR-IOV data structure in this list. The driver writes data to this structure to configure and enable SR-IOV and reads it to obtain SR-IOV status.

When the driver enables SR-IOV (it must be turned off after boot), the firmware creates virtual *functions* (in the PCI meaning). These functions are subsequently assigned unused PCI addresses from the same bus (leading to maximum 255 virtual functions). After the functions are enabled, they are configured (provided their memory mappings) by the kernel.

Subsequently, when the driver communicates with the device, it uses addresses from the device's mapped address range (see figure 5.2). The device only responds to requests with address within its range. When virtual functions are configured, each with its own address range, the physical device's logic needs to account for multiple ranges and process the request accordingly.

5.2 Design of driver with SR-IOV support

SR-IOV distinguishes two types of *functions*.

Physical function (or *PF*) is the original function of the SR-IOV device. It is available at system boot and is usually capable of device configuration. AT the very least, it is able to configure and enable SR-IOV through the SR-IOV capability structure.

Virtual Function (or *VF*) is a function created when SR-IOV is enabled through PF. This function usually has a limited or zero access to the device configuration, for it is meant to be assigned to a virtual machine, and as noted before, virtual machines should not affect each other's state. As a result, virtual function's abilities are usually limited

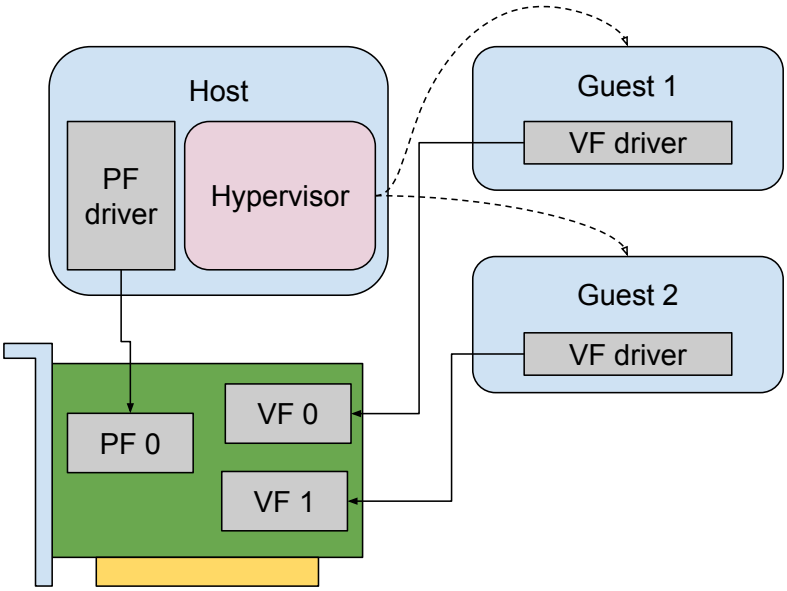


Figure 5.3: SR-IOV Driver Schema

to the performance-related operation (e.g. data transfer) and configuration that does not affect other virtual functions.

As a result, the SR-IOV system requires a PF driver and a VF driver. The VF driver provides the data transfer operations, while the PF driver provides data operations, configuration operations and SR-IOV configuration operations.

The host usually runs the PF driver, while the guests run VF drivers, see figure 5.3.

5.3 Design of firmware with SR-IOV support

The firmware of an SR-IOV device must address, in addition to the device's function, the following:

- SR-IOV configuration and enabling/disabling
- Providing configuration address space for virtual functions
- Separating transactions by virtual function number
- Multiplexing commands from virtual functions
- Limiting virtual function capabilities

The firmware can distinguish between virtual functions by the PCI address used in the respective transaction. This information can be used to provide proper result for the command or to block it, if it is configuration command issued by a virtual function.

Chapter 6

COMBO platform

The COMBO cards are programmable high-speed network cards developed in the *Liberouter*[1] project by CESNET[2] and, more recently, Netcope Technologies[3]. They work on 10 Gb, 40 Gb or 100 Gb Ethernet, and are capable of delivering full 100 Gbps of network traffic to software for processing.

In this work, we shall use the COMBO-100G2Q card. This card features two physical QSFP28 ports which can work in 4x10G mode (four 10Gb Ethernet links), one of them even in 1x100G mode (one 100Gb Ethernet link). We say that the whole card works in 100G1, 10G8 or 100G1-10G4 mode, based on the ports' configuration.

Mode	Port 0 mode	Port 1 mode	Total interfaces
100G1	1x100G	disabled	1
10G8	4x10G	4x10G	8
100G1-10G4	1x100G	4x10G	5

Table 6.1: COMBO card modes

The COMBO cards utilize multiple *DMA channels* – RX and TX queues accessible by software. Through the channels, packet classification is possible – firmware can implement a decision process to determine target DMA channel for each incoming packet.

6.1 Platform overview

The COMBO cards can be conceptually seen as an FPGA chip connected to the Ethernet ports and to the PCI-E interface, as seen in figure 6.1.

FPGA (*Field Programmable Gate Array*) is an electronic circuit, whose behavior is reconfigurable. FPGA is a middle way between an ASIC (very fast but single-purpose chip) and a generic processor (very flexible but slow chip). In the COMBO platform, the speed of the FPGA allows to process the extreme data speed, while its flexibility allows software-controlled changes in card behavior.

The FPGA consists of many logic blocks interconnected by programmable paths. When the FPGA is programmed, a sequence of bits (*bitstream*) is loaded into it, configuring what function the logical blocks shall perform and how the blocks shall be interconnected. The

[1] http://www.liberouter.org
[2] http://www.cesnet.cz
[3] http://www.netcope.com

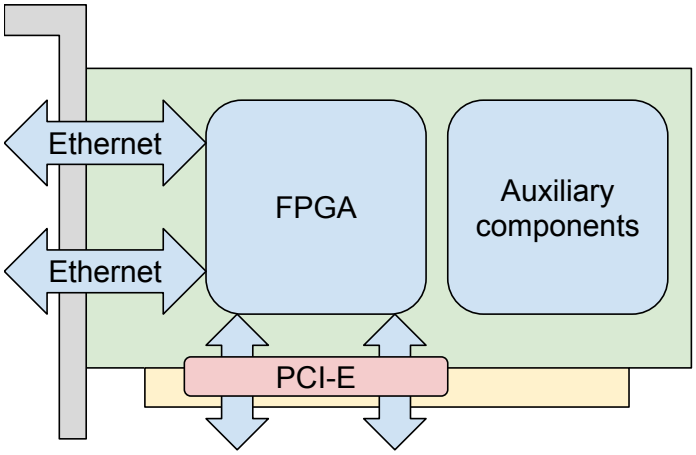


Figure 6.1: COMBO: High level overview

bitstream is usually generated from a specification in some hardware description language (VHDL, Verilog) in a design software tool from the chip's vendor. The design tool compiles the specification into a chip *design* – it analyzes the requirements from the specification, maps individual components to the logical blocks of the FPGA and places them on the chip in the manner allowing the whole *design* to run at the required frequency. The bitstream loaded into the card determines its behavior – it forms the card's firmware. In the rest of this work, we shall use the terms "design", "bitstream" and "firmware" interchangeably.

The COMBO designs consist of several common components, and an Application Core (see figure 6.2).

The application core is the packet processing unit of the firmware. It can process packets on the wire speed, allowing it to work as an accelerator. There are several existing application cores with different capabilities:

- **NIC** – no filtering, sends packets to software, can choose DMA channel (interface number or round-robin)
- **HANIC** – static packet filtering, can send packets to the software, crop them, send their Unified Headers to software, or drop them
- **SDM** – dynamic flow-based packet filtering, can send packets to the software, crop them, sned their Unfiied Headers to the software, or drop them

The IBUF and OBUF components are in control of a particular Ethernet port (in RX and TX direction, respectively). The RX_DMA and TX_DMA components are the DMA controllers of individual DMA channels. Note that it is possible to have varying number of RX and TX channels, because the RX channel is in no way tied to the TX channel.

6.2 Card Configuration

Hardware devices in the computer are controlled and configured by *software requests* – the processor reading and writing data to the device's control registers. The COMBO cards are capable of extensive configuration, utilizing many registers. This section covers the design of COMBO cards configuration system.

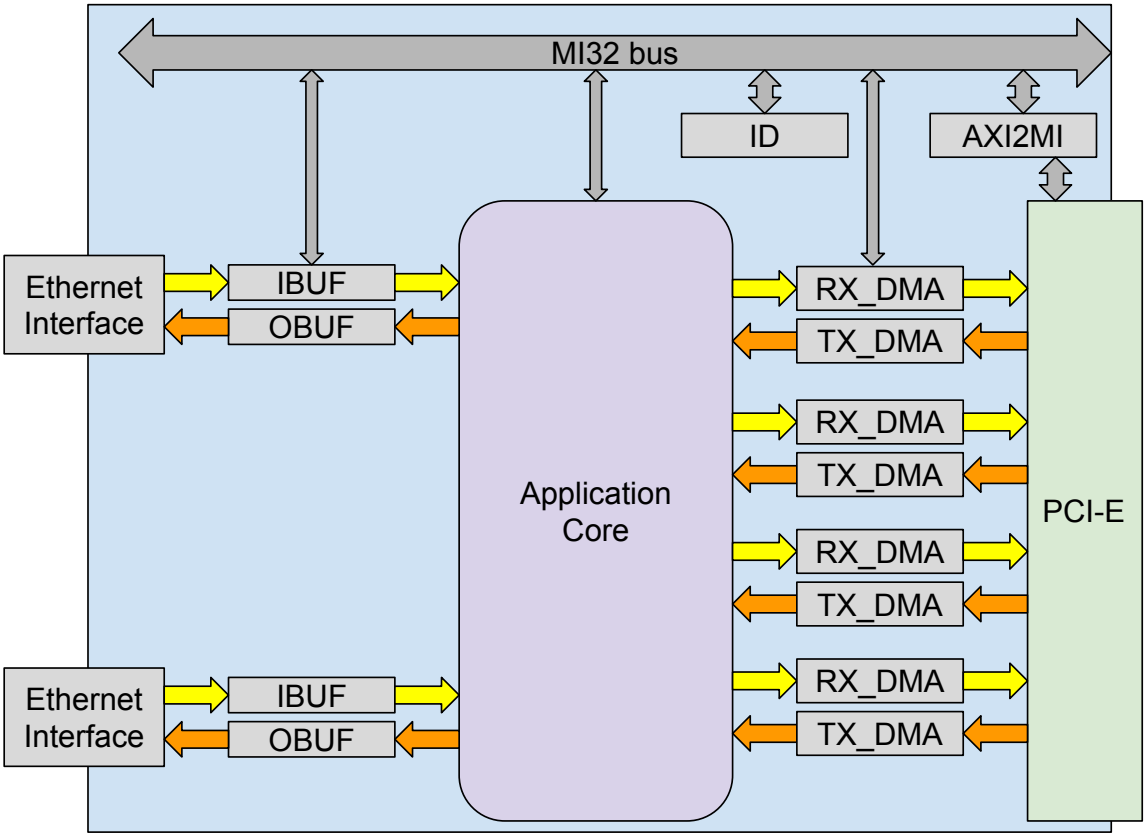


Figure 6.2: COMBO firmware data path components

When the system boots, each PCI device is assigned one or more continuous regions in the processor's address space (outside the addresses used by the actual RAM). The size of those regions is determined by the system through reading the BAR register, and upon the assignment, the base address of that region is written to the BAR. The system ensures that when the processor reads (or writes) data from the memory in the region assigned to a PCI device, this "request" shall be delivered to the correct device; the device shall respond appropriately.

The device can subtract the address in the BAR from the address in the request – basically, for the card, every request is accessing certain *offset* in certain BAR's address region.

6.2.1 Component addressing

The firmware consists of individual **components**. Each component that contains software-accessible registers must be placed to a unique offset in the card's address region, so that its registers may be uniquely addressed.

To achieve this, each component's designer must declare the component's *size* – the size of the memory region the component shall occupy. The firmware designer then enumerates all the components in the firmware and assigns to them base addresses in the firmware, taking into account their size so that their memory regions do not overlap. This assignment forms an *address space* of the firmware.

The firmware uses an internal bus called *MI32* to propagate the software requests inside the card. The bus consists of following signals:

`ADDR[31:0]`	Address (offset within the card)
`DRD[31:0]`	Data Read
`DWR[31:0]`	Data Write
`BE[3:0]`	Byte Enable (validity of individual bytes)
`RD`	Read (command)
`WR`	Write (command)
`ARDY`	Asynchronous Ready (command acknowledgment)
`DRDY`	Data Ready

When a request arrives to the card, the MI32 root component (*AXI2MI32*) asserts the `ADDR` signal and the `RD` (or `WR` and `BE`) signal. The MI32 bus contains address decoders which use the higher bits in the `ADDR` signal to activate the bus signals for the right destination component. The component then uses the lower bits in the `ADDR` signal to address the right register within itself and asserts `ARDY` (and `DRD` and `DRDY` if the command was *Read*) signal to acknowledge request completion.

Since the components only use the lower bits of the address for their internal addressing, they may be placed at any offset in the card, as long as this offset is aligned to a multiple of the component's size.

The component hierarchy differs between the firmwares. The software needs to know the offsets of individual components in the firmware it is working with, so every firmware is accompanied with an XML file called *design.xml*, which contains the component tree description in defined format. The applications parse it to know where the components they need to work with are located.

The *design.xml* contains XML nodes with the following structure:

```
<component cversion="1.0" version="2.4"
           name="IBUF" index="0" base="0x00008000" size="0x100">
    <comment>Input Buffer 0</comment>
</component>
```

Using the information in the card, the software can query for the existence (or count) of specific components in the firmware, learn their location (offset, base address) and access them correctly.

6.3 Data transfers

The main function of a network card is receiving and sending data. This section deals with high speed network data transfers between the software and the card.

6.3.1 SZE interface

When a new packet is received from the network, the network card stores it in the internal buffer and makes sure the packet data is retrieved by the operating system before it is overwritten by a new packet.

With the standard approach, the NIC fires an interrupt to inform the kernel about it. The kernel calls the NIC's driver to initialize DMA transfer to the kernel buffer, and when the packet is processed with the kernel network stack, it is finally copied to application buffer in a call like `recvmsg()`.

The approach is generic, flexible and allows the processor to work on other tasks when there are no packets available. However, the interrupt handling costs dozens or hundreds of CPU cycles, and the "kernel to application" copy doubles the memory utilization. This becomes a problem for higher Ethernet speeds.

Ethernet standard	**Data speed** 64B frames*	**Frame speed** 64B frames**	**Time / frame**	**Cycles / frame** 3.20 GHz processor
1 Gb/s	95.24 MB/s	1.488 Mpkts/s	672 ns	2150
10 Gb/s	952.38 MB/s	14.881 Mpkts/s	67.2 ns	215
40 Gb/s	3809.52 MB/s	59.524 Mpkts/s	16.8 ns	53
100 Gb/s	9523.81 MB/s	148.810 Mpkts/s	6.72 ns	21

Table 6.2: Packet processing time for high speed Ethernet

**Ethernet requires at least 12B of inter-frame gap and has 8B frame preamble, amounting to 20B per packet of transmitted, but unused data. Therefore, the theoretical maximal data speed for frames of a given size can be calculated as:*

$$DataSpeed = BitSpeed/8 * Framesize/(Framesize + 20) \tag{6.1}$$

***64B is the minimal size of an Ethernet frame, resulting in the highest possible frame speed*

With rising network speed, the interrupts become too slow and costly. Looking at table 6.2, it is clear that an interrupt (whose handling and even dispatching costs hundreds of microseconds [6]) cannot be fired for every received packet on higher Ethernet speeds and is unfeasible even for batch processing where an interrupt is fired for several packets.

As a result, the COMBO cards use (proprietary) technology called **Straight Zero Copy** (SZE). It consists of kernel modules (*szedata2, szedata2_cv3*) and user-space library *libsze2*. It does not use interrupts – instead, it allocates memory buffers and allows the network card to write data to these buffers through DMA. The buffers (areas) are arranged in the form of a single logical ring-buffer, where the card continuously writes new data and the software reads it. They update *Head* and *Tail* pointers respectively to inform the other party of the data / free space available in the ring-buffer. The disadvantage lies in the need for the application to constantly query the memory for new packets (so-called **polling**) and in the fact that SZE applications receive their packet as Ethernet frames, and cannot directly use the abstractions (like TCP sockets) provided by the kernel.

Today's computer systems employ multiple processor cores, allowing parallelization of tasks. COMBO cards utilize several *DMA channels* through which they send data to the processor. The SZE technology allocates independent ring-buffers for each channel, which allows multiple processor cores to perform packet processing simultaneously, reading from

individual channels. Steering individual packets to the DMA channels is performed by the card's firmware.

6.4 Software stack

The software stack of the COMBO platform consists of the card drivers, user-space libraries and configuration tools (see figure 6.4).

The card drivers (Linux kernel modules) manage the card as PCI device, and provide to the user-space device files to work with. The *combo6core* module provides functions common to all the COMBO cards. The *combov3* module, which handles the configuration accesses and provides access to the card components, creates file `/dev/combosixN`, where `N` is the card's number (starting from 0). Similarly, the *szedata2* module creates file `/dev/szedataIIN`, where `N` is the card's number. The binding module *szedata2_cv3* is used to provide the *szedata* module with the device-specific information.

The main userspace libraries, *libcombo* and *libsze2* provide functions for card configuration and data transfers, respectively. Most applications use at least one of these libraries, sometimes with common helper functions from *libcommlbr*.

User-space tools generally fall into two categories – configuration tools querying or modifying card status, and data transfer tools (whose name starts with `sze2`) designed for network traffic generation, receiving, transmitting, etc.

The COMBO platform is designed for custom applications, written on top of the *libsze2* library, working with the individual frames, without utilizing the kernel network stack. It is, however, possible to have the drivers create Linux network interfaces (*netdev*s) from individual DMA channel pairs (RX + TX channel). This allows to run standard network applications, albeit with limited performance. This is done by loading the *szedata2* module with parameters `no_eth=0` and `dma_channels=C1,C2,C3` where `C1`, `C2` and `C3` are the channels we want to use as *netdev*s.

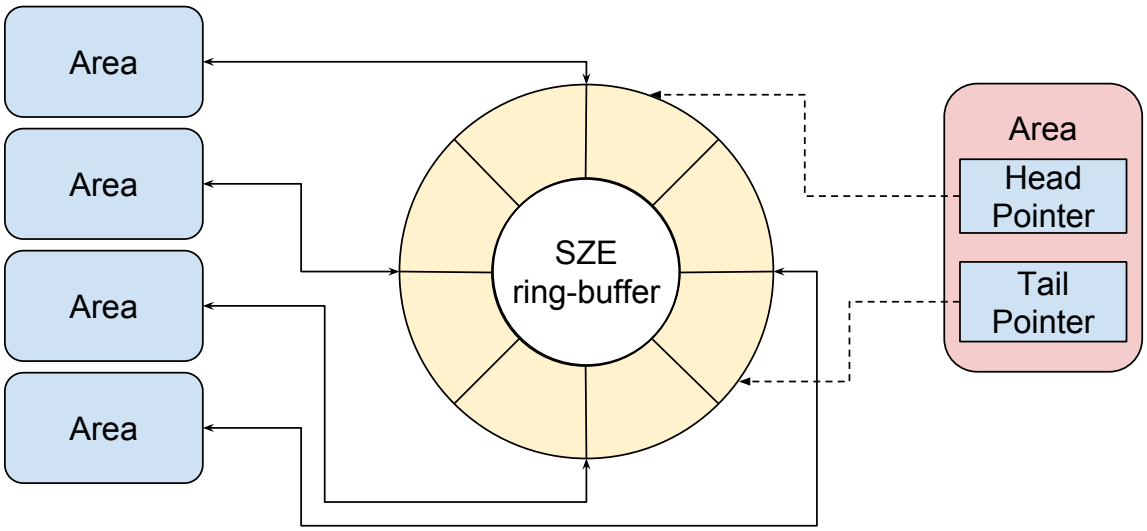


Figure 6.3: SZE ring buffer abstraction

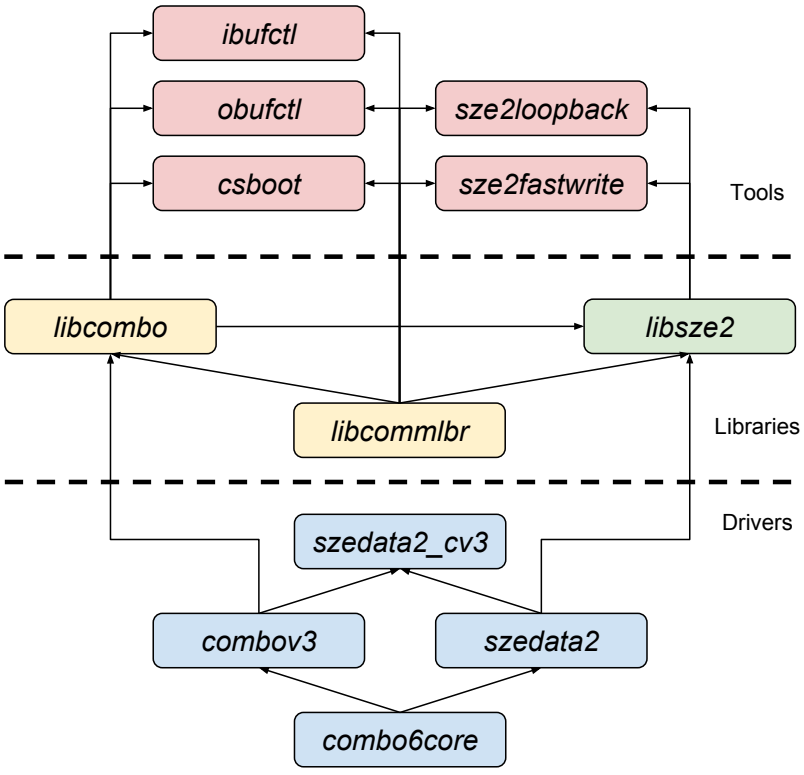


Figure 6.4: COMBO Platform software stack

Chapter 7

Design and implementation of SR-IOV support for COMBO cards

This chapter covers design and implementation of the support of SR-IOV technology for the COMBO platform. The goal is to add the support to the existing pieces of software and firmware (see section 6.4). With the SR-IOV support implemented, it shall be possible to create virtual functions in the COMBO card, assign them to virtual machines and use them in the VMs, achieving close to bare-metal performance.

To achieve the goal, the following tasks were defined:

1. the card must advertise its SR-IOV capability (see section 7.2),
2. the drivers must provide a way to create and manage virtual functions (see section 7.3),
3. the hardware resources in the card must be assigned to individual virtual functions (see section 7.4),
4. individual virtual function's hardware accesses must be distinguished and isolated from each other (see section 7.5),
5. and the VF driver must be created to be used in the VMs (see section 7.6).

In addition, the SR-IOV support requires the IOMMU unit to be turned on, which contradicts the requirements of standard COMBO installation due to the usage of PCI slot bifurcation. Section 7.1 covers both the IOMMU/bifurcation problematics and the implemented solution.

The firmware support is added to the code in the *fwbase* repository hosted by CESNET, the software support is added to the code in the *netcope-common* repository hosted by Netcope Technologies, a.s.

7.1 IOMMU, bifurcation and PCI-E endpoints

The PCI Express bus consists of multiple separated serial point-to-point links (or *lanes*) which connect the physical slot to the PCI Express Root Complex, interconnecting the CPU and other devices. An individual physical slot can contain 1, 4, 8 or 16 such lanes (the slots are accordingly marked *x1*, *x4*, *x8*, *x16*).

The current version of the PCI-E bus, present in the server motherboards and supported by Xilinx Virtex7 PCI-E IP Core, is PCI Express version 3.0, which offers transmission speed **8.0 GT/s**. GT/s stands for *GigaTransfers per second*. Since each lane is serial, it transfers a single bit at a time, the effective data transfer speed of a single lane is nearly 8 Gb/s[1].

In order to achieve full 100 Gbps throughput to software, an *x16* lane or an equivalent has to be used, since 8 lanes per 8 Gbps can only transfer nearly 64 Gbps. However, the Xilinx PCI-E IP Core used in the COMBO platform can only create an x8 endpoint. This has been overcome by instantiating two Xilinx PCI-E IP Cores in the COMBO platform and enabling **PCI Slot Bifurcation** feature of the motherboard (it essentially turns the *x16* physical slot into two *x8* slots).

This allows the desired 100 Gbps transfers, but from the system's view, it turns the card into two separate PCI-E devices – primary and secondary. The COMBO drivers perform only a minimal initialization of the secondary device, all the work is done through the primary device. The secondary device is only used for SZE data transfers. Therefore, the system sees two separate devices both writing the data to the same memory area (SZE buffers).

However, with IOMMU on, every device can only transfer data to the memory region that was allocated to it through `dma_alloc_coherent()` or other kernel function. The reason is that the IOMMU maps devices' memory accesses. When using IOMMU, the devices are expected to use *bus addresses* instead of *physical addresses* and, similar to the MMU unit, the IOMMU establishes mapping $(device, busaddr) \rightarrow physaddr$ which is then used for the device's memory accesses. The bus address is returned to the driver on successful DMA memory allocation call (like `dma_alloc_coherent()`), which also sets up the IOMMU mapping.

So, why is the IOMMU needed at all? A running virtual machine is seen as a user process by the host system. Like any other process, it works with virtual memory – the memory addresses the VM uses are virtual addresses and are translated to physical addresses by the MMU unit. When such a VM is assigned a PCI device and starts a DMA transfer with it, it programs the device with the address of the DMA buffer in the memory. But, since the VM uses virtual memory, the programmed address (which the virtual machine considers physical memory address) might not match the real physical address where the data should be transferred.

As a result, for direct device assignment to a VM, the device's memory accesses must undertake the same mapping that is used for the VM. Hence, for an SR-IOV environment, the IOMMU must be present and turned on.

The COMBO firmwares can be built using a single endpoint only. This is done by setting the `DOUBLE_PCIE` variable to `„0"`

```
set DOUBLE_PCIE "0"
```

in the main configuration file in the firmware application's top directory (`fwbase: applications/nic/100g2/top/Vivado.tcl`)

The change also requires removing unused signals and pins from the whole design (see the patches on the CD).

[1] PCI Express version 3 uses 128b/130b encoding, which lowers the effective speed by cca 1.5%

7.2 Advertising SR-IOV Capability

The SR-IOV capability is a data structure in the PCI device's *configuration space*[2] used by a PCI device to inform the system that the device is SR-IOV capable (see section 5.1).

In the COMBO firmwares, the PCI-E communication, including managing the *configuration space*, is handled by *Virtex-7 FPGA Gen3 Integrated Block for PCI Express*; an IP core (prebuilt block of FPGA logic) provided by Xilinx, Inc.

7.2.1 Xilinx PCI-E IP Core

Xilinx allows licensed users of their products to incorporate the PCI-E IP core into their firmwares. Using Xilinx's IDE, *Vivado*, it is possible to configure the PCI-E IP core and add it to the firmware. The configuration creates a file containing the PCI-E IP core settings (referred to as *wrapper file*). Since the COMBO firmware uses the IP core, the file is already present; in the *fwbase* repository, the file path is

```
ndk
└── common
    └── comp
        └── external
            └── ip_cores
                └── pcie
                    └── virtex7
                        └── general
                            └── pcie3_7x_0_wrapper.v
```

When the IP core is configured in *Vivado*, a new wrapper file is generated in the directory, from which Vivado was started. Its path is

```
<PROJECT>.srcs/sources_1/ip/pcie3_7x_0/synth/pcie3_7x_0.v
```

where `<PROJECT>` is the name of the Vivado project (usually "combo100g2_core").

In the following section, I will cover the BAR registers, since they are set up in the IP Core configuration process, and later the configuration of the IP core for SR-IOV support and its integration to the COMBO firmware, which will be done by partial replacement of the old wrapper file with the one produced during configuration.

Base Address Registers

Base Address Registers (BARs) are registers in the PCI device's configuration space which are used by the system to determine device's memory requirements and to store the address on which the system will map the device. Since PCI devices are accessed via memory read/write instructions, each device must be mapped somewhere in the processor's logical address space.

Upon system boot, when PCI devices are probed, the BIOS or the kernel will write 1's to their BARs and read back the value. The read value determines the *size* of the BAR – the size of the address space it requests to be assigned. Subsequently, the system allocates address space region of required size and assigns it to the BAR (writes the lowest address to the BAR). From this moment, accesses to the addresses in this region shall reach the device, and the device shall use the Address value in the request and the Base Address in the BAR to determine the access's offset within itself.

[2] Don't mix with COMBO configuration accesses

In COMBO firmwares, this is how configuration accesses are performed. The memory accesses are transformed - first to the AXI bus by the PCI-E core, then to the MI32 bus by the AXI2MI component. Therefore, the size of the BAR0 must encompass all the components on the MI32 bus. Basic NIC components are located on offsets up to `0xF000`, but some special ones go up to `0x880010` and the application core employed in some designs is usually located at `0x2000000`. That requires the BAR to be at least `0x4000000` (64 MiB) large. Recently, another BAR was added to the firmware for experimentation. With SR-IOV, using this second BAR is discouraged, because the memory requirements shall grow due to the BARs for individual VFs, and the system may run out of assignable address range (an error known as "not enough MMIO resources").

The VFs are designed to work with the basic components only. As a result, their BARs can be as small as `0x10000` (64 KiB).

7.2.2 PCI-E IP core configuration

For the IP core configuration, you need Vivado Design Suite. The work was performed in Vivado 2016.3, although other versions may work as well.

For IP core configuration, it is possible to either create a new project in Vivado, or generate one containing the COMBO firmware and open it in Vivado. To achieve the latter, navigate to the *top* directory of your desired application (e.g. `applications/nic/100g2/top`), edit file *Vivado.tcl*, change the line

```
set SYNTH_FLAGS(PROJ_ONLY) „0"
```

to

```
set SYNTH_FLAGS(PROJ_ONLY) „1"
```

and run `make` to generate Vivado project file *combo100g2_core.xpr*.

In Vivado GUI, locate the IP Catalog in the left pane and when it opens, locate *Virtex-7 FPGA Gen3 Integrated Block for PCI Express* under

```
Standard Bus Interfaces
└─PCI Express
  └─Virtex-7 FPGA Gen3 Integrated Block for PCI Express
```

Double clicking (or choosing "Customize IP" from the context menu) will open a window with IP customization. In this window, we shall modify certain fields to configure the IP core for SR-IOV support properly. The screenshots of the window with appropriate fields can be found in the Appendix.

Following tables list changes to be performed in individual tabs.

Table 7.1: Tab "Basic"

Field	New value	Old value	Note
Mode	Advanced	Basic	Enable advanced configuration options
Lane width	X8	X1	Enable x8 PCI-E lane
Maximum Link Speed	8.0 GT/s	2.5 GT/s	Enable PCI-E gen. 3 link speed

Table 7.2: Tab "Capabilities"

Field	New value	Old value	Note
SRIOV Capability	*Checked*	*Unchecked*	Advertise SR-IOV capability to the system

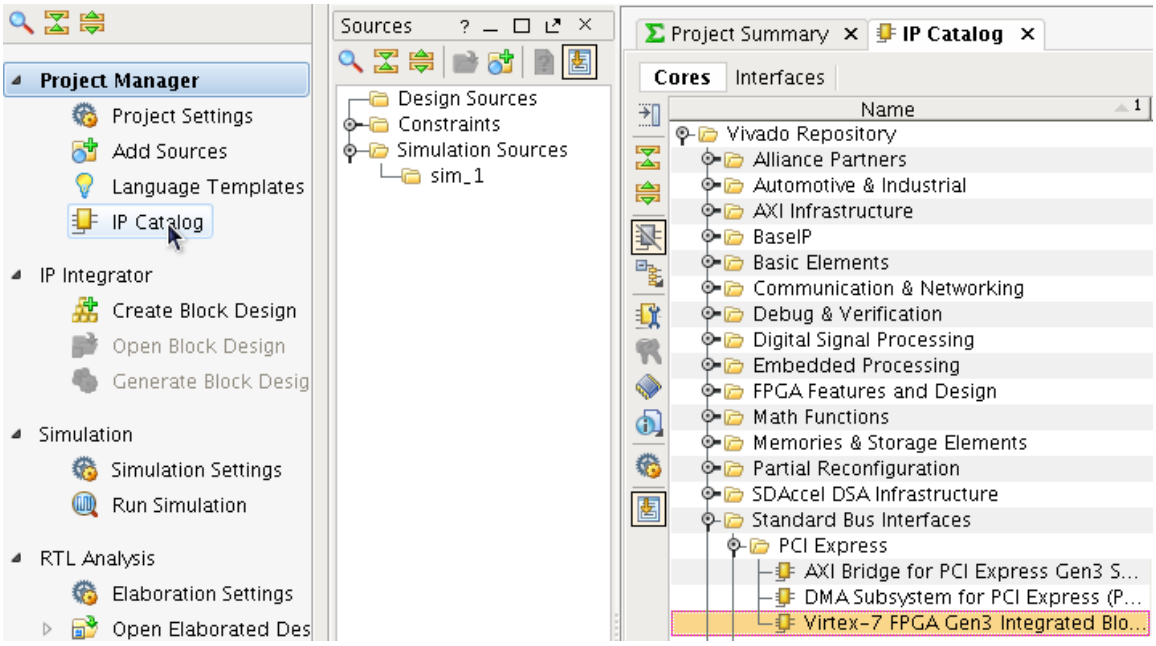


Figure 7.1: Selecting PCI-E IP Core in Vivado 2016.3

Table 7.3: Tab "PF0 IDs"

Field	New value	Old value	Note
Vendor ID	1B26	10EE	Use INVEA-TECH vendor ID
Device ID	C0C1	7038	Use Combo-100G2 device ID
Subsystem Vendor ID	1B26	10EE	Use INVEA-TECH vendor ID
Subsystem Device ID	0000	0000	
Base Class Value	02	05	Network Controller class
Sub Class Value	00	04	Ethernet controller subclass

This is for compatibility with the COMBO firmware values only. Can be left unchanged if caution is applied when merging wrapper files.

Table 7.4: Tab "PF0 BAR"

Field	New value	Old value	Note
Bar0: 64-bit	*Checked*	*Unchecked*	Enable 64-bit BAR
Bar0: Size Unit	Megabytes	Kilobytes	
Bar0: Size Value	64	4	Use 64MB BAR0

This is for compatibility with the COMBO firmware values only. Can be left unchanged if caution is applied when merging wrapper files.

Caution: *Enabling BAR2 may conflict with SR-IOV support, because the system may not have enough MMIO resources (memory available for PCI devices) to allocate big BAR0, big BAR2 and small BARs for VFs.*

Table 7.5: Tab "SRIOV Config"

Field	New value	Old value	Note
Number of PF0 VF's	6	0	Enable maximum (6) VFs

Table 7.6: Tab “PF0 SRIOV BARs”

Field	**New value**	**Old value**	**Note**
Bar0: 64-bit	*Checked*	*Unchecked*	Enable 64-bit BAR
Bar0: Size Value	64	2	Use 64KB BAR0

After the changes are confirmed with “OK”, Vivado offers to generate the IP core “out-of-context”. Confirming this, IP core synthesis is asynchronously started. When it finishes, the new wrapper file is created and ready to be used.

Running a `diff` on the two wrapper files shows many differences. We will modify the configuration lines, following the line

```
pcie3_7x_0_pcie_3_0_7vx #(
```

We shall apply those modifications, which relate to the SR-IOV capability, other capabilities and BAR sizes, namely:

- .ARI_CAP_ENABLE(„TRUE")
- .PF0_AER_CAP_NEXTPTR('H140)
- .PF0_ARI_CAP_NEXTPTR('H200)
- .PF0_BAR0_APERTURE_SIZE('B10011)
- .PF0_BAR2_APERTURE_SIZE('B00000)
- .PF0_BAR2_CONTROL('B000)
- .PF0_DEV_CAP_EXT_TAG_SUPPORTED(„TRUE")
- .PF0_DPA_CAP_NEXTPTR('H200)
- .PF0_DSN_CAP_NEXTPTR('H200)
- .PF0_LTR_CAP_NEXTPTR('H200)
- .PF0_PB_CAP_NEXTPTR('H200)
- .PF0_RBAR_CAP_NEXTPTR('H200)
- .PF0_SRIOV_BAR0_APERTURE_SIZE('B01001)
- .PF0_SRIOV_BAR0_CONTROL('B110)
- .PF0_SRIOV_CAP_INITIAL_VF('H0006)
- .PF0_SRIOV_CAP_TOTAL_VF('H0006)
- .PF0_SRIOV_FIRST_VF_OFFSET('H0040)
- .PF1_SRIOV_BAR0_APERTURE_SIZE('B01001)
- .PF1_SRIOV_BAR0_CONTROL('B110)
- .SRIOV_CAP_ENABLE(„TRUE")
- .VF0_PM_CAP_NEXTPTR('H90)
- .VF1_PM_CAP_NEXTPTR('H90)
- .VF2_PM_CAP_NEXTPTR('H90)

- `.VF3_PM_CAP_NEXTPTR('H90)`
- `.VF4_PM_CAP_NEXTPTR('H90)`
- `.VF5_PM_CAP_NEXTPTR('H90)`

7.3 Virtual Function management

SR-IOV virtual functions management is rather limited. The user may only specify the number of requested virtual functions (henceforth designed K) and, upon success, first K virtual functions are enabled. Disabling is done by specifying the K as zero.

The management is done in the PF device driver (the **combov3** module). It must provide a way for the user to specify K and use the kernel-provided functions:

```
pci_enable_sriov(pcidev, K);
pci_disable_sriov(pcidev);
```

for the actual enabling/disabling.

The newer kernels (starting with kernel 3.8) allow management through a `/sys` file, namely `/sys/bus/pci/devices/0000:03:00.0/sriov_numvfs` for a PCI device located at `03:00.0`. Here is an example of virtual function management through the `/sys` file:

Listing 7.1: Example of `/sys`-based VF management

```
# cat /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
0

# lspci | grep 1b26
03:00.0 Ethernet controller: Device 1b26:c2c1

# echo 6 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs

# cat /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
6

# lspci | grep 1b26
03:00.0 Ethernet controller: Device 1b26:c2c1
03:08.0 Ethernet controller: Device 1b26:0000
03:08.1 Ethernet controller: Device 1b26:0000
03:08.2 Ethernet controller: Device 1b26:0000
03:08.3 Ethernet controller: Device 1b26:0000
03:08.4 Ethernet controller: Device 1b26:0000
03:08.5 Ethernet controller: Device 1b26:0000
```

In order for this method to work, the device driver module must define an SR-IOV configuration function, which handles VF management (it can perform driver-specific initialization and then call `pci_sriov_enable()`). The configuration function must have prototype

```
int sriov_configure(struct pci_dev *pcidev, int num_vfs);
```

and the `sriov_configure` field of the driver's `struct pci_driver` structure must be filled with a pointer to this function. Otherwise the attempts to write or read the `/sys` file end with error.

Older kernel versions, including kernel 2.6.32 present in CentOS 6, do not have the `sriov_configure` field in the `struct pci_driver` structure, so this method cannot be used. As an alternative, I chose to use a `/proc` file (called `/proc/combo-sriov`) which could be created during module loading and removed during its unloading, and whih performed the same function (except that there could only be one file, which would cause problems i multiple COMBO cards were present).

The module checks kernel version and through conditional compilation (see listing 7.2) it uses the `/sys` file version, if the kernel is new enough to support it, otherwise the `/proc` version.

Listing 7.2: **combov3** module SR-IOV configuration support checking

```
#define HAS_SRIOV 1

#ifdef HAS_SRIOV
#if LINUX_VERSION_CODE >= KERNEL_VERSION(3,10,0)
#define NEW_SRIOV 1
#else
#define OLD_SRIOV 1
#endif
#endif
```

7.4 Management of hardware resources

The hardware contains a limited number of resources, and the simultaneous access from multiple VFs would cause problems, if the same resource was used more than once.

This led to the decision to use the **NIC** firmware, which can work in a *pass-through* mode, that is, it is built with the same amount of physical interfaces (IBUF/OBUF pairs) and DMA channels (RX and TX DMA controllers) and its core only forwards packets from n-th IBUF to n-th DMA channel, and from n-th DMA channel to the n-th OBUF, as depicted in figure 7.2.

This setup splits the components into *groups* independent of each other (IBUF 0's state cannot affect RX_DMA 1's state and so on), so it is possible to assign different groups to the different VFs (as shown on figure 7.3).

In order for a VF to be able to transmit and receive data, it must access the DMA channel controllers (in order to set up the SZE addresses and pointers). In order to be able to manage its physical port, it needs access to the IBUF and OBUF of that port. And finally, the drivers and the software need access to some information about the card, provided by the ID component.

So, concluded, here are the components that must be provided to a VF in order to make it a self-managed data transferring NIC:

- IBUF (component controlling ingress physical interface)
- OBUF (component controlling egress physical interface)
- RX_DMA (component controlling RX DMA channel)

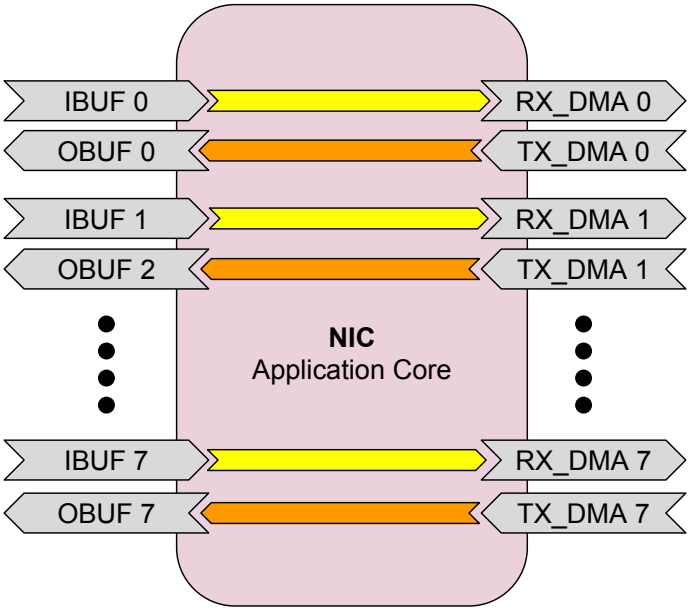


Figure 7.2: NIC Application Core scheme

- TX_DMA (component controlling TX DMA channel)
- ID (component providing basic information about the card and the firmware)

Since the ID component is used read-only, it is possible to share it among the VFs. As for the other components, it is possible to assign n-th *group* to the n-th VF.

7.5 VF component access

We have learned before that the components inside firmware are controlled by memory operations (*Read, Write*) on certain addresses within the physical card's assigned memory range.

In order to access certain register in certain component, the software must know the offset of the component within the card, and the offset of the register within the component. While the latter is the internal knowledge of the software tool *(a tool working with a component X ought to know the component X's registers' offsets)*, the former is subject to change among different designs and is provided to the software tool through the *design.xml* file.

The VFs shall, by design, work with only a subset of the components present in the firmware. Therefore it is needed to design **what** components they shall use (see section 7.4) and **how** shall they access them.

The naive approach would be removing the unused components from each VF's virtual machine *design.xml* files. However, this is dangerous, as it allows the possibility for the VFs to access components not assigned to them (be it by misconfiguration or malicious intent). The other disadvantage is that the VFs are not uniform – each requires a slightly different *design.xml* file, as each's components lie on different offsets.

These problems can be solved by adding a firmware component, which will manage the VFs' accesses to the other components. Two possible approaches were suggested – access

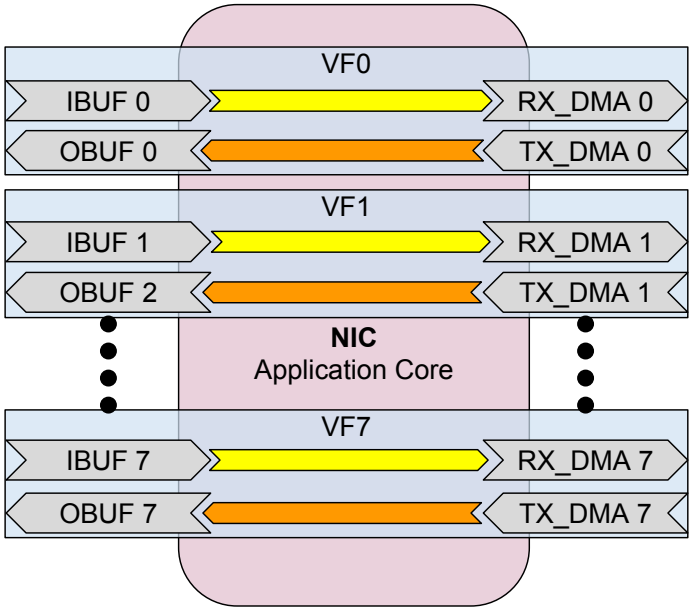


Figure 7.3: COMBO components assignment to the VFs

filtering and access mapping. For both approaches, the component must be aware of the individual components' VF assignment.

With access filtering, the component shall block the accesses outside the memory regions of the components assigned to the requesting VF. This approach solves the security problem, however, the non-uniformness issue remains, as each VF needs to use different offsets, hence needs a different *design.xml*. It could be partially solved by providing the VF number to the VM and the software, and utilizing it to choose the right file or add proper offsets to the data in it.

Access mapping is an approach which takes inspiration in the memory mapping of the userspace addresses, and the MMU unit. The basic idea is to create a *virtual* address space (represented by *design.xml*) for the VFs and then map accesses to this virtual space to the real components, using the VF number to find the right one. Similar to the pages in the regular virtual memory context, only a portion of the address, which determines the component, would be mapped, while the lowest part of the address, representing an offset *within* the component, would remain intact.

The access filtering only solves the security issue, while the access mapping allows for uniform VFs and may perform access filtering too. As a result, we have chosen to use **access mapping**.

7.5.1 VF virtual address space

For the VF' successful operation in the access mapping environment, the VF address space needs to be designed – the VF must be presented with a *design.xml* file containing the components accessible to it, placed on the offsets the VF software shall use.

Each VF shall have access to 5 components: the *ID* component (shared, read-only), and the *IBUF*, *OBUF*, *RX_DMA* and *TX_DMA* components with the same index as is the VF's number (e.g., VF0 shall have access to IBUF0).

Component	Offset	Size
ID	0x0	0x1000
IBUF 0	0x8000	0x200
OBUF 0	0x9000	0x100
RX_DMA 0	0xC000	0x200
TX_DMA 0	0xD000	0x200

Table 7.7: Widely used locations for certain components

Although most of the software works with *design.xml*, there are tools that use hard-wired constants for component location (offsets). This works because in nearly all the designs, most of the components are placed on the same offset. The table 7.7 lists standard locations of the components relevant for the design.

To maintain the compatibility and not to break the software relying on these offsets, we have decided to design the VF address space, as if the VF was a regular firmware with but a single interface, and a single DMA channel, that is, we have placed the components of the VF on the same location as shown in the table 7.7. The mapping for individual virtual functions s depicted in figure 7.4.

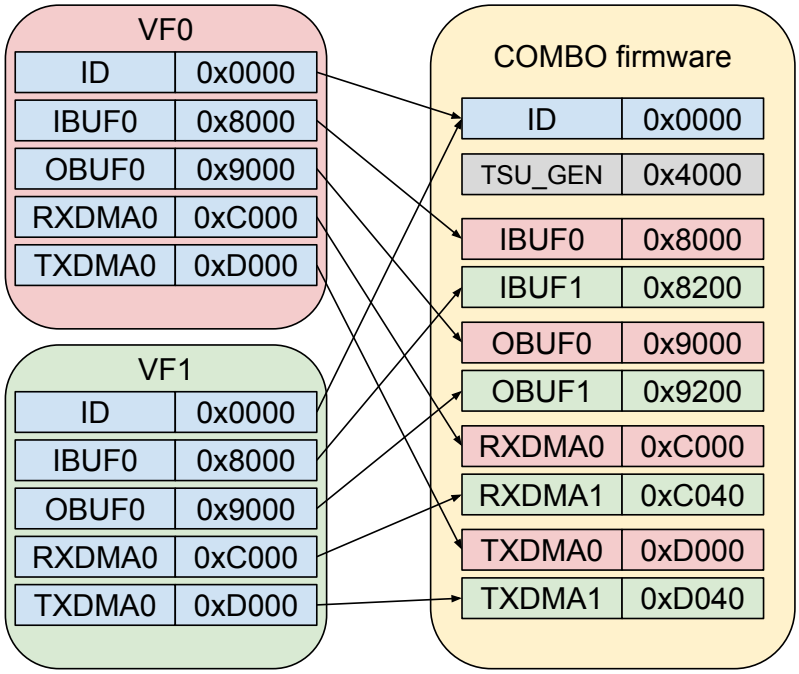


Figure 7.4: COMBO component mapping for individual virtual functions

7.5.2 MI_VFT component

Mapping of the configuration accesses is implemented in the MI_VFT component of the COMBO firmware. The component (depicted on figure 7.5) is placed on the MI32 bus and performs address mapping (and sometimes data modification) for the MI32 transactions.

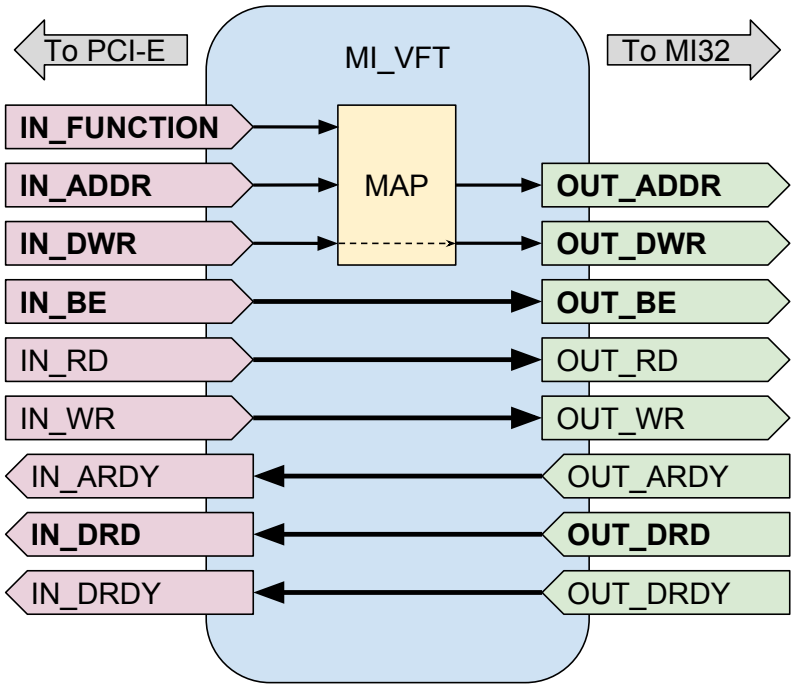


Figure 7.5: MI_VFT component architecture

The component only maps accesses from the virtual functions. Using the `IN_FUNCTION` signal (whose value is 0 for the PF and $64 + i$ for i-th virtual function), it changes the address signal as described below.

It uses the fact that the IBUFs, OBUFs and individual DMA channel controllers have a constant offset between them, so the part of the address space can be interpreted as an array, which is indexed by the virtual function number.

For example, IBUF0's base address is `0x8000` and offset between IBUFs is `0x200`. Therefore an access to an address in the range [`0x8000`, `0x81FF`) by virtual function 3 should be mapped to the range [`0x8600`, `0x87FF`), preserving the offset.

The range [`0x8000`, `0x81FF`), expressed in binary, is

```
1000 0000 0000 0000
 ...
1000 0001 1111 1111
```

Therefore, this can be tested by comparing the upper 7 bits to a binary constant `0b1000000`. Since the MI32 bus uses 32-bit addressing, there are four more zero bytes above, so in the VHDL syntax, the comparison constant is `X„00008" & „000"`.

The component code for such mapping is listed below.

```
OUT_ADDR <= X"0000" & "1000" & IN_FUNCTION(2 downto 0) & IN_ADDR(8 downto 0)
            when IN_ADDR(31 downto 9) = (X"00008" & "000");
```

The ID component is mapped flat:

```
OUT_ADDR <= IN_ADDR when IN_ADDR(31 downto 8) = X"000000";
```

Since the *design.xml* file contains only the main RX_DMA and TX_DMA controllers (the controllers for individual channels are located through known offsets), a register in the ID component is used to determine the count of DMA channels present in the design. In order to provide this information to the virtual functions as well (the original read-only register cannot be used), a new register was added to the ID component, and a fixed mapping

```
OUT_ADDR <= X"00000078" when IN_ADDR = 64;
```

ensures that the virtual functions use the new register.

In specific cases, the component modifies written data as well. When the individual DMA channels are configured, they must learn their virtual function number, so that they can tag data sent to the Xilinx PCI-E IP Core and the proper PCI device is used for them. This is done by writing the virtual function number to the DMA controller register. This could be done in the software layer, but it would require the software writing the data to be aware of its virtual function number, which we tried to avoid.

As a result, the virtual function number is added to the data by the MI_VFT component when the address corresponds to a DMA controller's control register. The mapping follows.

```
OUT_DWR <= IN_FUNCTION & IN_DWR(23 downto 0)
           when (IN_ADDR = X"0000C000" or IN_ADDR = X"0000D000")
           else IN_DWR;
```

7.6 Virtual function driver

The virtual function driver is designed to operate in a virtual machine, controlling one of the virtual functions. Standard COMBO driver was analyzed and modified to work as a VF driver.

From the system's point of view, the virtual function behaves as a standard COMBO card, with limited configuration capabilities. The driver already has a system of card's capabilities (to support various cards from the COMBO family), so a new capability, `COMBO3_CAP_VIRTFN`, was added to hold the information whether or not a PCI device is a virtual function.

The driver code uses the capability flag to disable the code which works with the PCI-E bus or sets interrupts (functions like `pci_set_master()` or `request_irq()`. It also inhibits code which works with the components not present in the virtual function design.

The virtual function uses a different Device ID than all the COMBO cards, therefore regular drivers do not recognize the virtual functions as COMBO cards. This is useful in the host, because the COMBO drivers are usually loaded there and if they recognized the virtual function Device ID, they would start controlling it. For this reason, the support for this particular Device ID is enabled in the driver by setting `VM_DRIVER` to `1` in the `kernel/drivers/combov3/cv3.c` file. It is also advised that the virtual function driver is compiled without physical function SR-IOV support. This is achieved by setting `HAS_SRIOV` to `0` in the same file. The lines

```
#define HAS_SRIOV 1
#define VM_DRIVER 0
```

should be changed to

```
#define HAS_SRIOV 0
#define VM_DRIVER 1
```

for the VF driver.

7.7 Implementation summary

This section lists the modifications to the overall code-base which were necessary for SR-IOV support implementation.

Firmware modifications

All changes are in the *fwbase* repository.

- Changed the PCI-E IP Core configuration
  - `fwbase:ndk/common/comp/external/ip_cores/pcie/virtex7/general/pcie3_7x_0_wrapper.v`
- Implemented the MI_VFT component
  - `fwbase:ndk/100g1/src/comp/mi_vft/mi_vft_arch.vhd`
- Added the DMA channel count register for virtual functions
  - `fwbase:ndk/common/comp/base/misc/id32/id_comp.vhd`
- Switched the design to use only one PCI-E endpoint
  - `fwbase:applications/nic/100g2/top/Vivado.tcl`

Driver modifications

All changes are in the *netcope-common* repository.

- Added VF management (enable, disable) through `/sys` (new kernels) and `/proc` (old kernels)
  - `sources/drivers/kernel/drivers/combov3/cv3.c`
- Added "VIRTFN" capability and used it to disable PCI operations for virtual functions
  - `sources/drivers/kernel/drivers/combov3/cv3.c`
- Implemented per-channel DMA controller configuration
  - `sources/drivers/kernel/drivers/combov3/sze2cv3.c`

Chapter 8

Performance Evaluation

The goal of the performance tests was to evaluate performance (in terms of RX and TX speed) of the virtual function NICs in QEMU/KVM virtual machines.

The tools used were

- data transfer tools (`sze2fastwrite, sze2loopback`) from the *netcope-common* package for transmitting and receiving data
- Bash built-in `time` for elapsed time measurement

The tested machine was installed and configured according to the appendix A. Its COMBO card was connected to another COMBO card (on a different physical machine, running *Netcope* standard NPC firmware), which was used as a traffic generator. They were connected using 8 10Gb/s Ethernet cables. The setup is depicted on figure 8.1.

For each test, the environment was set up (see below), then the data transfer tool was started (with its execution time measured using `time`) and kept running for approximately 10 seconds. Then it was terminated using the SIGINT signal and the values of processed packets and elapsed real time were recorded.

Each test was performed in the **"single"** environment (where only the measured tool was running) and in the **"full"** environment, where equivalent tools were launched on all

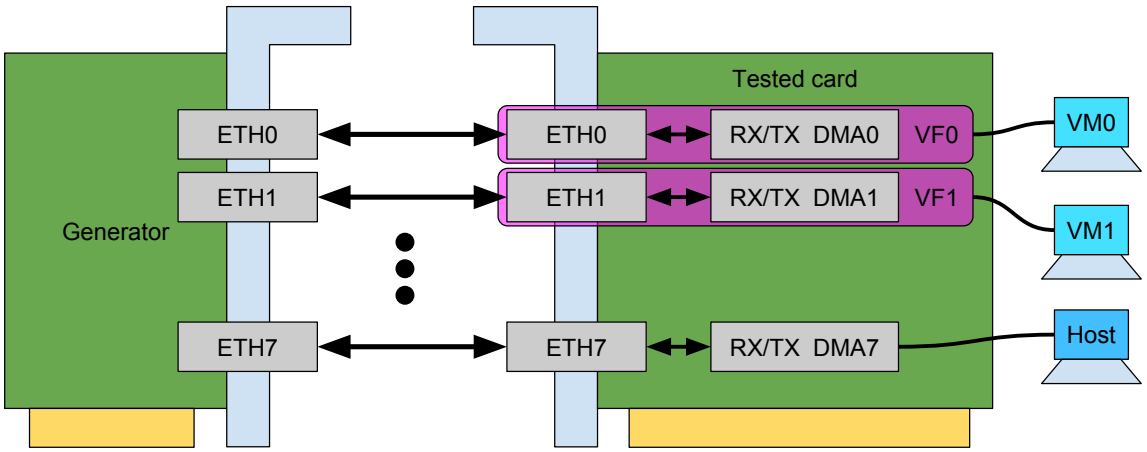


Figure 8.1: Performance Evaluation card setup

Direction	Packet size	Mode	Time [s]	Packets	Packet speed [Mpps]	Data speeed [Mb/s]
RX	64 B	Single	13.686	203437342	14.865	7610.691
TX	64 B	Single	14.457	214972222	14.870	7613.321
RX	1500 B	Single	12.529	10291947	0.821	9857.400
TX	1500 B	Single	11.355	9279071	0.817	9806.468
RX	64 B	Full	13.402	147449030	11.002	5633.033
TX	64 B	Full	11.080	127183333	11.479	5877.276
RX	1500 B	Full	13.843	7997814	0.578	6932.968
TX	1500 B	Full	10.622	5972739	0.562	6747.905

Table 8.1: COMBO with SR-IOV performance in virtual machines

the virtual machines and on the unused DMA channels 6 and 7[1], so that 8 tools were run simultaneously.

For the RX direction evaluations, the generator card generated 10 Gbps of traffic (with specified fixed packet size) on each interface.

Tested packet sizes were 64 and 1500 B – the smallest and nearly largest allowed packet sizes (see [1, p.108]) for regular Ethernet.

The results of the evaluations are presented in table 8.1.

8.1 Performance Analysis

Theoretical maximum speed (data throughput) of an Ethernet link is dependent on the size of the transmitted packets. The Ethernet standard [1] defines an 8B preamble transmitted before every Ethernet frame and mandates that there is a gap between individual frames transmission. This gap (*interpacket gap*) must be at least 96 bits (12 B) long. This means that every transmitted packet has 20 B overhead. Therefore, theoretical maximum speed S_n for transmitting packets with size n bytes on an Ethernet link with speed S is calculated as

$$S_n = S * \frac{n}{n + 20} \tag{8.1}$$

With this knowledge, we can calculate theoretical maximum speed S_{64} and S_{1500} for 64B and 1500B packets on 10Gb Ethernet link, shown in table 8.2.

Packet size	S_n [Mbps]
64	7619.05
1500	9868.42

Table 8.2: Theoretical maximum speed for certain packet sizes on 10Gb Ethernet

We can see that in single mode, the performance very closely reaches the theoretical maxima.

The "full" mode is expected to have worse performance due to the fact that the design only runs on a single *x8* PCI-E lane (see section 7.1). The single lane provides nearly

[1] the Xilinx PCI-E IP Core only supports up to 6 virtual functions

64 Gbps throughput, which corresponds to nearly 8 Gbps per DMA channel. The SZE technology adds some overhead on the PCI-E bus, since some of the transactions are used to update the SZE pointers (see section 6.3.1) and each packet transferred by SZE is 8B aligned and prefixed with an 8B or 16B SZE header.

So, assuming the SZE protocol overhead is 2%, theoretical maximum speed F_n of a single channel in the "full" configuration for packet size n is calculated as (assuming PCI-E bus throughput for one channel to be F)

$$F_n = F * 0.98 * \frac{n}{(\lceil \frac{n}{8} \rceil * 8) + 16} \tag{8.2}$$

With 8 processes running the same application, we shall assume the bus throughput to be divided equally among them – the F then equals approximately 7877 Mbps[2].

This allows us to calculate theoretical maximum speed F_{64} and F_{1500} for 64B and 1500B packets in the "full" mode, described in table 8.3.

Packet size	F_n [Mbps]
64	6175.57
1500	7617.89

Table 8.3: Theoretical maximum speed for certain packet sizes on 7.877Mbps SZE channel

We can see that the measured values for the "full" mode are approximately 10% below the theoretical maximum. This can be caused by higher SZE protocol overhead (since the 2% are an estimate) or by overhead caused by multiple virtual machines contending for the PCI-E bus access.

Generally, the SR-IOV support has enabled the virtual machines to transfer data at nearly wire-speed, with the drawback of the single PCI-E endpoint being supported, limiting the overall system throughput.

[2]Less then 8000 due to the 128/130 encoding

Chapter 9

Conclusion

In the first part of this work, the problematics of network input/output operations virtualization was analyzed and described. The work provided motivation and reasons for adopting virtualized environments, and discussed requirements to provide high-speed virtual network cards in these environments. Standard network traffic processing in Linux kernel was described, documenting its complexity and overhead hindering high performance network applications and software-based virtualization.

Individual virtualization technologies, namely *emulation*, *paravirtualization* represented by the *virtio* technology, and *full virtualization* represented by the *SR-IOV* technology, were described and compared to each other in terms of performance and flexibility. The SR-IOV technology was described more closely.

Based on the information from the first part, support of the SR-IOV technology has been designed and implemented for the COMBO hardware platform. The design included modifications to the COMBO card firmware, as well as the card drivers.

Firmware changes included reconfiguration of the card's PCI Express IP Core and adding a component for mapping accesses from virtual functions. Enabling the IOMMU unit (required for SR-IOV) resulted in limiting the firmware to a single PCI-E endpoint, which has lowered the platform's maximal throughput.

The driver changes included virtual function management using `/sys` or `/proc` file, recognizing the virtual function as a specific COMBO card, and extending the driver's capability system to prevent PCI-E related operations being performed for the virtual function.

Performance of the system was evaluated with multiple virtual machines receiving and transmitting data simultaneously. The results have shown close to wire-speed performance for "single virtual machine" test cases, "full throughput" test cases (where all the machines communicated simultaneously) demonstrated lower performance, since the combined data throughput has reached the limits of the single PCI-E endpoint.

This work can be expanded further by

- finding a way to run the SR-IOV design with both PCI-E endpoints (could be achieved by hard splitting the DMA channels to the endpoints or through tweaking the IOMMU configuration with IOMMU groups in modern kernels)
- designing and implementing support for multiple DMA channels in a single virtual function

Bibliography

[1] IEEE Standard for Ethernet. *IEEE Std 802.3-2015*. 2015.

[2] Chiosi, M.; Clarke, D.; Willis, P.; et al.: *Network Functions Virtualisation – Introductory White Paper*. White paper. ETSI. 2012.
Retrieved from: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[3] Han, B.; Gopalakrishnan, V.; Ji, L.; et al.: *Network function virtualization: Challenges and opportunities for innovations. IEEE Communications Magazine*. vol. 53, no. 2. Feb 2015: pp. 90–97. ISSN 0163-6804.
doi:10.1109/MCOM.2015.7045396.

[4] Jonathan Corbet, G. K.-H., Alessandro Rubini: *Linux Device Drivers*. O'Reilly Media. Third edition. 2005.
Retrieved from: https://lwn.net/Kernel/LDD3/

[5] Jones, M. T.: *Virtio: An I/O virtualization framework for Linux*. 2010.
Retrieved from: https://www.ibm.com/developerworks/library/l-virtio/

[6] Kim, I.; Moon, J.; Yeom, H. Y.: Timer-Based Interrupt Mitigation for High Performance Packet Processing. In *In Proc. 5th International Conference on HighPerformance Computing in the Asia-Pacific Region, Gold*. 2001.

[7] Kutch, P.: *PCI-SIG SR-IOV Primer*. Technical report. Intel Corporation. 2011. [Online; accessed 15-Dec-2016].
Retrieved from: http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html

[8] Lowe, S.: *What is SR-IOV?* [Online; accessed 15-Dec-2016].
Retrieved from: http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/

[9] Müller, P.: *Virtualizace platformy x86*. Bachelor thesis. Brno University of Technology. 2008.

[10] Rosenblum, M.; Waldspurger, C.: *I/O Virtualization. Queue*. vol. 9, no. 11. November 2011: pp. 30–39. ISSN 1542-7730.
Retrieved from: http://doi.acm.org/10.1145/2063166.2071256

[11] Russell, R.: *Virtio: Towards a De-facto Standard for Virtual I/O Devices.* SIGOPS Oper. Syst. Rev.. *vol. 42, no. 5. 2008: pp. 95–103. ISSN 0163-5980. doi:10.1145/1400097.1400108.*
Retrieved from: http://doi.acm.org/10.1145/1400097.1400108

[12] *Intel® VMDq Technology Overview. Technical report. Intel Corporation. 2008. Retrieved from:* http://www.intel.com/content/www/us/en/virtualization/vmdq-technology-paper.html

[13] *Virtual Machine Device Queues. [Online; accessed 15-Dec-2016]. Retrieved from:* http://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/io-acceleration-technology-vmdq.html

[14] *Wikipedia: Hardware virtualization — Wikipedia, The Free Encyclopedia. 2016. [Online; accessed 15-Dec-2016]. Retrieved from:* https://en.wikipedia.org/w/index.php?title=Hardware_virtualization&oldid=719994632

Appendix A

Installation

This chapter provides the information on the environment in which the work was performed, as well as requirements that must be met in order to run a COMBO card and a virtual machine, as well as installation and configuration steps needed to repeat the process.

A.1 Hardware setup

The COMBO cards are designed to be run on a GNU/Linux system on 64-bit x86 processors.

The cards require an x8 or x16 PCI-E slot. When x16 slot is used, the hardware must support and enable *PCI Slot Bifurcation (x8x8)* which allows the system to work with the x16 endpoint as with two x8 endpoints. Note that the bifurcation is required for the regular operation of the COMBO cards only, the firmware with SR-IOV support does not require it. It is also advised to turn off HyperThreading, if the processor supports it.

For KVM-based virtualization with SR-IOV support, the machine must support hardware virtualization (Intel's VT-x or AMD's AMD-V), must contain the IOMMU unit (which is called VT-d by Intel) and must support the SR-IOV technology itself. Usually, these features are optional and can be enabled in BIOS.

For running virtual machines, it is advisable to have more RAM than all the virtual machines' total allocated RAM size, and a multi-core processor with at least 1 core per virtual machine.

The following list describes reference hardware configuration.

- ASRockRack EPC612D4I
- Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
- 4 * 8 GB Kingston DDR4 RAM @2133 MHz
- BIOS
  - HyperThreading (HT): **OFF**
  - SR-IOV Support: **ON**
  - Virtualization (VT-x): **ON**
  - IOMMU (VT-d): **ON**
  - *See figure A.1 for BIOS settings screenshots*

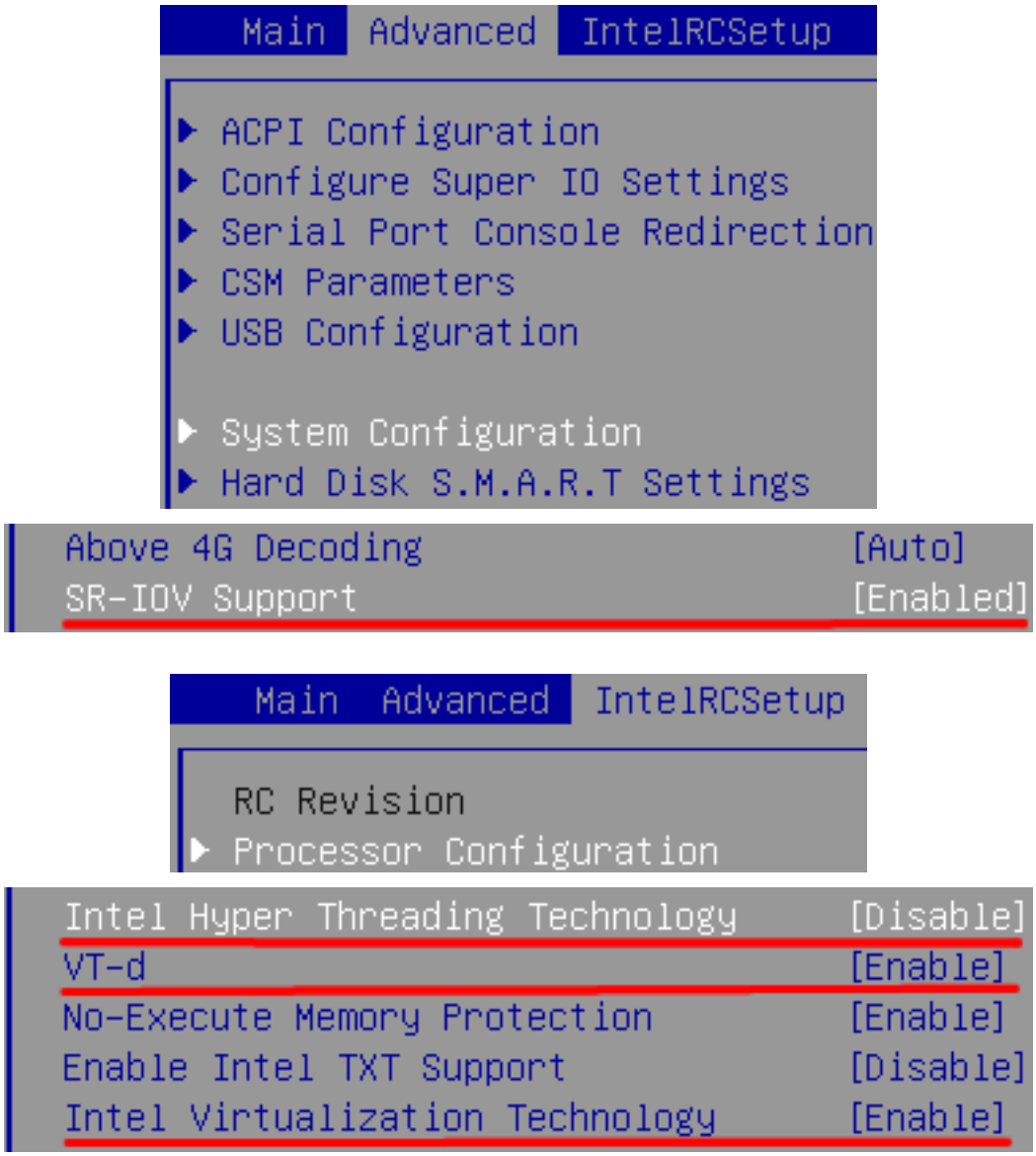


Figure A.1: Setting the required items in AsRock BIOS

A.2 Operating system and software

At the time of writing, the operating system supported by the COMBO software stack was CentOS 6[1]. *Note that CentOS 7 has a different kernel configuration which is not compatible with the COMBO software stack.* The system was installed from CentOS 6 Minimal DVD ISO file and subsequently updated.

On the vanilla, up-to-date system, the `epel-release` package was installed in order to get the `dkms` package needed for COMBO drivers. After that, the `netcope-common` package (from Netcope Technologies[2]), version 3.0.2, containing the standard COMBO card drivers, as well as basic software tools for card management, was installed. The package installs several other packages for dependencies, notably `kernel-headers`, `kernel-devel`, `dkms`, `gcc` and `make`, which allow the drivers to be built on the target machine for the running kernel.

To summarize, following steps need to be performed:

1. Update the system (`yum -y update`)
2. Reboot it (`reboot`)
3. Install the `epel-release` package (`yum -y install epel-release`)
4. Obtain and install the `netcope-common` package
   (`yum -y install ./netcope-common-3.0.2-1.x86_64.rpm`)

A.3 Setting up virtualization

For virtualization (creating and management of the virtual machines), the QEMU/KVM + Libvirt solution was chosen, since this solution is native to Linux, allows to utilize the hardware acceleration and allows PCI device pass-through required for SR-IOV.

For the KVM setup, the information were retrieved from CentOS wiki[3], specifically the command for installing the required packages:

```
yum -y install @virt* dejavu-lgc-* xorg-x11-xauth tigervnc \
libguestfs-tools policycoreutils-python bridge-utils
```

and the command to enable the *libvirtd* daemon and restart the system:

```
chkconfig libvirtd on; shutdown -r now
```

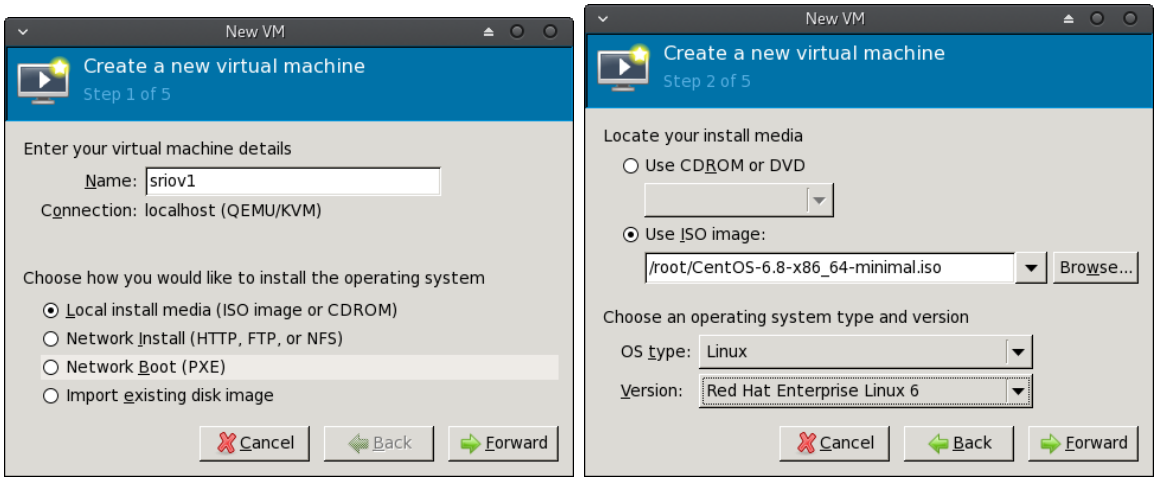
The other commands from the **Host Setup** were found unnecessary, since the packet forwarding is enabled automatically by *libvirtd* and bridging was not needed.

For creating the virtual machines, the graphical manager, **virt-manager**, was used. Screenshots of the procedure are depicted on figure A.2.

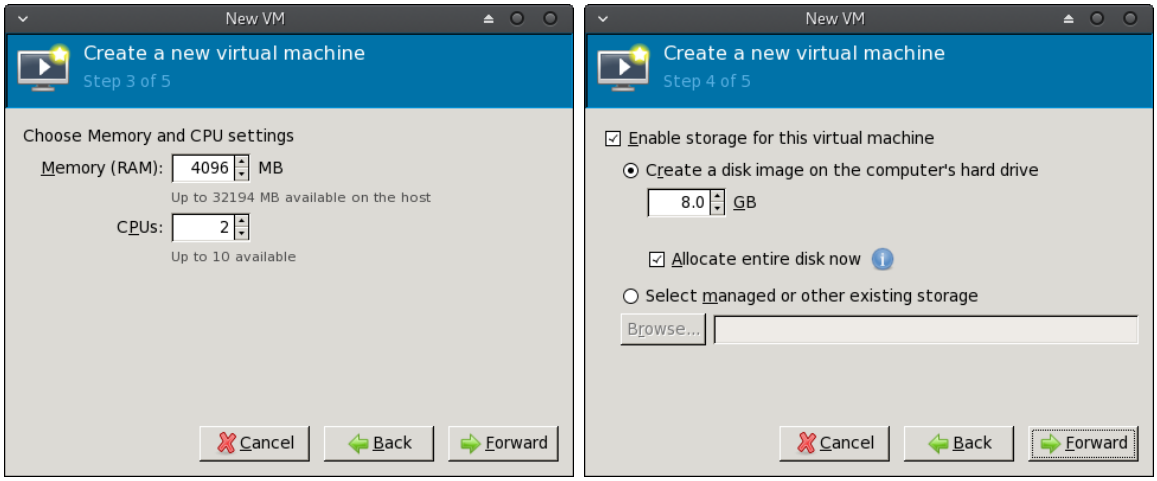
Note that without any further settings, the **virt-manager** installation creates default network and virtualized NIC running in NAT mode for the guest. This means the guest has network capabilities.

The guest operating system is CentOS 6 as well, and the installation is the same as described in section A.2.

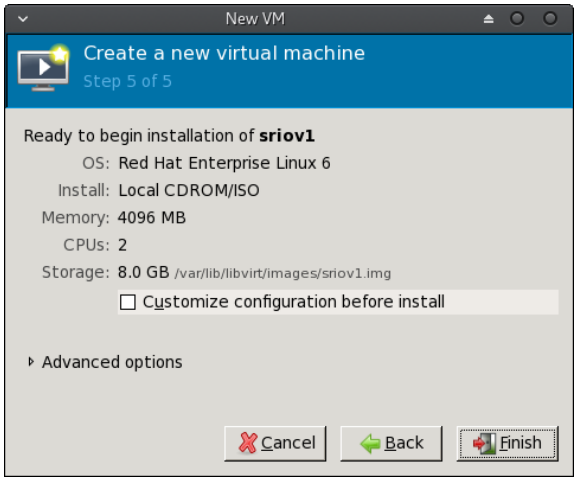
[1] http://www.centos.org
[2] http://www.netcope.com
[3] https://wiki.centos.org/HowTos/KVM



1 **2**



3 **4**



5

Figure A.2: Creating new virtual machine in **virt-manager**

A.4 Installing the SR-IOV support

For SR-IOV support, you need to build SR-IOV capable COMBO design, compile both PF driver in the host and VF driver in the guest, set up their design files and use the hypervisor to assign the virtual functions to a virtual machine.

Building SR-IOV firmware

In order to build the SR-IOV firmware, you need complete *fwbase* repository, the *fwbase* patches from the CD, *Vivado* design tool and COMBO build environment[4].

The patches should be applicable to the master branch of the repository, commit `eb5dc74bffeaeab98ae8997619a1a758bdce593a`.

When the patches were applied, running `make` in the `applications/nic/100g2/top` directory of the repository should invoke Vivado (provided the COMBO buil environment is set) and build the design (produce files *combo100g2_core.bit* and *combo100g2_core.msc*).

When the firmware is built, the produced firmware files (either of them) may be copied to the machine and booted to the card using the `csboot` tool from the *netcope-common* package.

```
csboot -f100 /root/combo100g2_core.bit
```

Building SR-IOV drivers

In order to build the SR-IOV drivers, you need the *drivers* directory of the *netcope-common* repository, target kernel headers and standard Linux build and development tools (`gcc, autoconf, automake` and others). All required software (except the drivers themselves) is installed through dependencies for the *netcope-common* package.

The drivers are available on the CD int he *netcope-common* directory (the patches in the *patches* directory were included to demonstrate the work that I actually did).

In the `netcope-common/sources/drivers` directory, invoke

```
# aclocal
# autoconf
# ./configure
# make
```

to build the drivers.

When building the VF driver, it is advised to turn off SR-IOV support by changing the defined value `HAS_SRIOV` in the `kernel/drivers/combov3/cv3.c` to 0.

```
#define HAS_SRIOV 0
```

For the drivers to become active, you must load their kernel modules using `insmod`. Note that the *netcope-common* package contains the default drivers, so you will need to remove them first. The following sequence of commands, issued from the *drivers* directory, removes current COMBO driver modules and loads the built ones.

```
# rmmod szedata2_cv3 szedata2 combov3 combo6core
# insmod ./build/drivers/base/combo6core.ko
# insmod ./build/drivers/combov3/combov3.ko
# insmod ./build/drivers/szedata2/szedata2.ko
# insmod ./build/drivers/combov3/szedata2cv3.ko
```

[4] Contact the CESNET TMC department

Installing design files

The design files for the PF and VF are available on the CD as *design_pf.xml* and `design_vf.xml`. The tools expect them symlinked to the locaation specified by the firmware's SW ID and Text ID (`0x41c10700` and `NIC_100G2_10G` for this firmware).

Running

```
mkdir -pv /usr/share/mcs/NIC_100G2_10G
ln -s /root/design_pf.xml /usr/share/mcs/NIC_100G2_10G/0x41c10700
```

installs the PF design file located in `/root`. Without the design file installed, the tools will report missing or invalid file. You can check installation correctness by running

```
csid -s
```

This applies to the host and to the guest in the same way, except that in the guest, the VF design file should be installed. Installing the PF design file in a guest machine may lead to system hangup (due to potential invalid accesses).

Assigning virtual functions to the virtual machine

After you have successfully built an SR-IOV firmware and installed SR-IOV drivers and design file on the host, you can enable N virtual functions by writing the N value to the `/proc/combo-sriov` file (or the appropriate `/sys` file if you have new enough kernel, see section 7.3).

You can verify success by running `lspci`, which should list N newly created `Netcope Technologies, Device 0000` or `1b26:0000` devices. Also, `dmesg` should report no errors. You should note the PCI identification of the virtual functions for the actual assigning step described below.

When you have virtual functions available, you may assign them to a virtual machine created int the hypervisor. This can be done (in QEMU/KVM + Libvirt) either through editing the virtual machine XML file (`virsh edit`) or using `virt-manager` GUI (see figure A.3). Either way, you need to specify the PCI ID of the function you are assigning.

When the assignment is finished, you may start the virtual machine. If this fails, it is likely you do not have IOMMU turned on.

After the machine has started, if you have built the drivers on it and loaded the modules, you should be able to use the virtual COMBO card in the virtual machine. You can verify it by running

```
# csid -s
```

and test the installation with

```
# obufctl
# obufctl -Ae1
# sze2fastwrite
^C
# obufctl
```

If more than 64MB of data were transmitted, the installation can be considered successful.

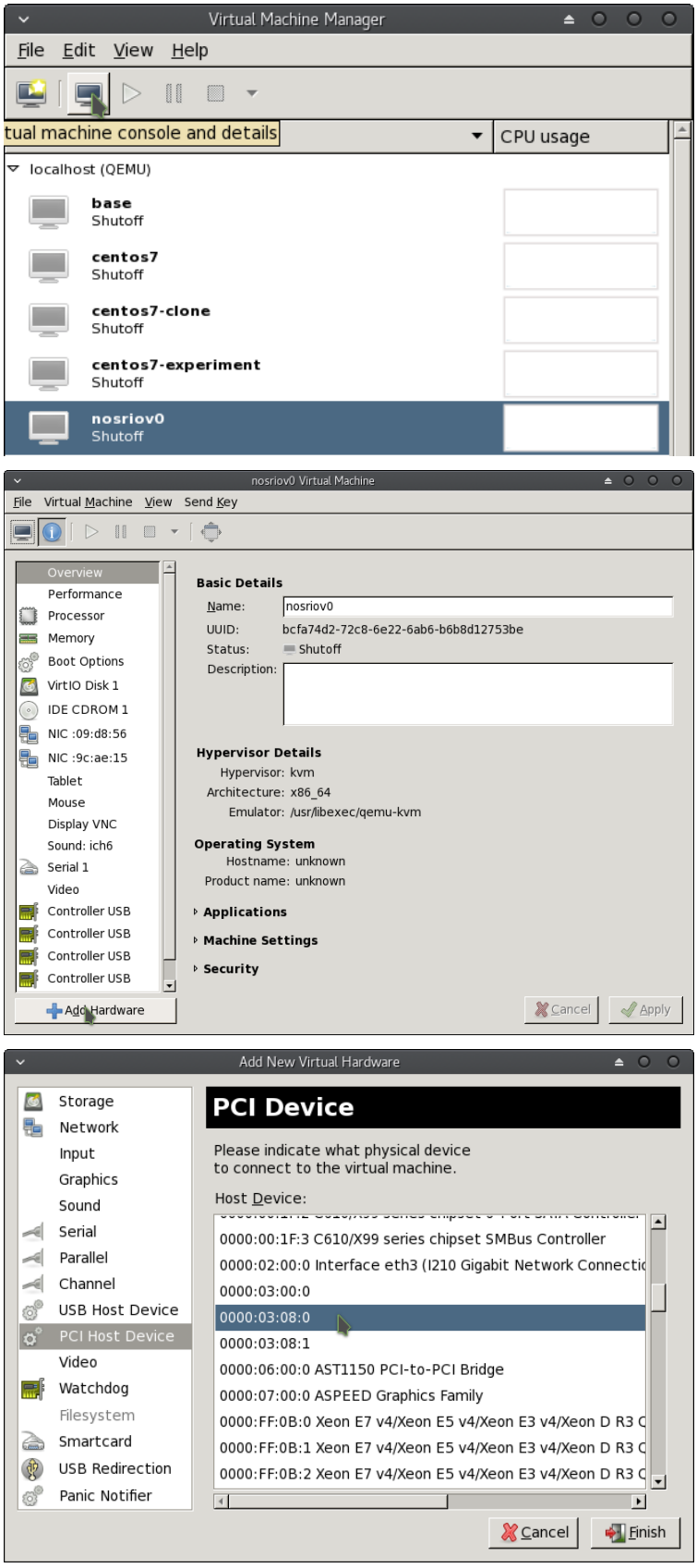


Figure A.3: Assigning virtual function to virtual machine