BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

I/O VIRTUALIZATION IN NETWORKING

VIRTUALIZACE I/O OPERACÍ V OBLASTI POČÍTAČOVÝCH SÍTÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MARTIN PEREŠÍNI**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MARTÍNEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Perešíni Martin, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Virtualizace I/O operací v oblasti počítačových sítí**
I/O Virtualization in Networking
Kategorie: Operační systémy
Zadání:

1. Study and understand current I/O virtualization technologies.
2. Get acquainted with the NDK platform developed by the CESNET association.
3. Design a suitable way to virtualize I/O operations for the NDK platform.
4. Implement the proposed method and verify its functionality on available hardware.
5. Evaluate the results and discuss the possibilities of continuing the project.

Literatura:

- According to the instructions of the supervisor.

Při obhajobě semestrální části projektu je požadováno:

- Fulfillment of items 1 to 3 of the assignment.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Martínek Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 25. února 2020

// Abstract
There are many different reasons for companies and organizations to invest in virtualization today, but it is probably safe to assume that financial motivation is number one on the list. Virtualization can save a lot of money. This thesis deals with the problem of I/O virtualization in the network environment in order to keep pace with this trend. The ultimate goal of this thesis is to develop working I/O virtualization software drivers that operate with FPGA-accelerated cards, thus enhancing their potential even more and saving the operational costs of big data centers. Their main benefits should be re-usability (flexibility) in a virtualized environment with the smallest possible performance loss. The theoretical part deals with current trends in I/O virtualization, technologies such as virtio, vhost, SR-IOV, VFIO and mediated devices. The practical part of this thesis suggests two ways of addressing this problematic. The first is to use software-emulation virtio technology. The second is based on the hybrid paravirtualization VFIO-mdev technology. Both approaches have different benefits in terms of performance and device manageability. Each solution's use case has its own drawback, like the complexity of the solution and the problematic integration into the system. The desired goals were achieved and manifested in the final form of the prototype driver `nfb_mdev`.

Abstrakt
Existuje veľa rôznych dôvodov pre spoločnosti a organizácie, prečo by mali investovať do virtualizácie. Asi najväčší dôvod je finančná motivácia, pretože nasadenie virtualizácie môže ušetriť nemálo peňazí. Táto práca sa zaoberá práve problémom virtualizácie I/O operácií v sieťovom prostredí. Cieľom práce je tvorba softvérových ovládačov pre I/O virtualizáciu, ktoré by mohli pracovať s hardvérovo akcelerovanými sieťovými kartami. Hlavným prínosom ovládačov by mala byť použiteľnosť a čo najmenšia strata prenosového výkonu vo virtualizovanom prostredí. Pred popisom finálnych detailov ovládačov je však potrebné uviesť potrebné teoretické základy. Teoretická časť sa zaoberá súčasnými trendami vo virtualizácii I/O, technológiami ako sú virtio, vhost, SR-IOV, VFIO a mdev. V praktickej časti sú navrhuté dva spôsoby riešenia problému. Prvým je použitie technológie virtio (emulácia softvéru). Druhé je založené na technológii VFIO-mdev (hybridná paravirtualizácia). Pokiaľ sa jedná o výkon a konfigurovateľnosť zariadení, oba prístupy majú rôzne benefity. Tieto riešenia majú aj svoje nevýhody, ako je zložitosť riešenia a náročnosť integrácie do systému. Požadované ciele boli úspešne dosiahnuté vo forme prototypu ovládača `nfb_mdev`.

Keywords
virtualization technologies, I/O device virtualization, peripherals, software drivers, computer networks, network cards, KVM, QEMU, PCI-Express, IOMMU, PASID, Intel VT-d, Netcope, NDK, FPGA, virtio, vhost, SR-IOV, VFIO, mediated devices - mdev

Kľúčové slová
virtualizačné technológie, virtualizácia I/O zariadení, periférne zariadenia, softvérové ovládače, počítačové siete, sieťové karty, KVM, QEMU, PCI-Express, IOMMU, PASID, Intel VT-d, Netcope, NDK, FPGA, virtio, vhost, SR-IOV, VFIO, mediated devices - mdev

Reference
PEREŠÍNI, Martin. *I/O Virtualization in Networking.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Martínek, Ph.D.

I/O Virtualization in Networking

Declaration

I hereby declare that that this Master's thesis was prepared as an original work by the author, solely independently under the leadership of Mr. Ing. Tomáš Martínek, Ph.D. Further information about this problematic was provided to me by CESNET employees within the Liberouter research group, especially insights from Mr. Ing. Martin Špinler. I have listed all the literary sources, publications and other resources that I used in writing this thesis.

. . . .
Martin Perešíni
June 3, 2020

Acknowledgements

In this way I decided to thank my supervisor Mr. Ing. Tomáš Martínek, Ph.D. and my colleague Mr. Ing. Martin Špinler. I would also like to thank my mother and father for providing the means to continuing my studies and for their support. I would also like to thank my brothers and friends for making my time more pleasant.

Contents

1 Introduction **4**

2 Concepts and technologies associated with virtualization **6**
2.1 The art of virtualization 6
2.2 Virtualization basics in the Linux operating system 8
2.3 PCI-Express bus from a software engineer's point of view 10
2.4 IOMMU and direct memory access 17
2.5 Netcope Development Kit (NDK) platform 20

3 Technologies of network interface I/O virtualization **23**
3.1 Software I/O virtualization – I. Generation 23
3.2 Direct I/O device assignment – II. Generation 25
3.3 Scalable I/O virtualization – III. Generation 27

4 Designing new device drivers **32**
4.1 Vhost software driver 32
4.2 VFIO-mdev software driver 34

5 Implementation **37**
5.1 `npp_vhost` driver 37
5.2 `mdev_nfb` driver 39

6 Evaluation **53**
6.1 Evaluation 53
6.2 Problems with the DMA, `PASID` experiments 58

7 Conclusion **61**

Bibliography **63**

A Storage Medium **66**

B Handler of PCI extended capabilities in VHDL **67**

C Samples of `nfb_mdev` driver implementation, source code **70**

D Testing the implementation **73**

List of Figures

2.1 Scheme of common computing architecture compared to a virtualized computing architecture that features an abstract virtualization layer. 7
2.2 Linux based host system architecture virtualized with KVM/QEMU. 9
2.3 (a) Hierarchical details of libvirt architecture. (b) Level view of libvirt implemented in a virtualized environment. 10
2.4 Scheme of a typical PCI subsystem including the devices and their connection to the buses and PCI bridges. 11
2.5 PCI memory layout preview. 12
2.6 The PCI configuration space. 13
2.7 An example of a whole PCI configuration space with its capabilities. 15
2.8 An simplified example of PASID concept. 16
2.9 A composition of a PCI PASID Extended Capability Structure. 17
2.10 A schematic representation of MMU and IOMMU memory management units. 17
2.11 DMA concepts w/o virtualization, w/ virtualization and why it is necessary to introduce an IOMMU unit. 18
2.12 Concepts of an IOMMU addressing within a domain/group, device and process (by `PASID`). 19
2.13 NDK Platform for the development of hardware-accelerated network applications over FPGA technology. 20
2.14 The photography of COMBO-100G2Q and NFB-200G2QL cards developed on the NDK platform. 21
2.15 Software component scheme of NDK platform. 22
2.16 A hierarchical structure of the NDK platform software package including drivers, library and card tools. 22

3.1 (a) Scheme of a software I/O virtualization. (b) Scheme of a software I/O virtualization using a virtio technology. 24
3.2 (a) Virtio implementations in the Linux kernel. (b) Virtio transport mechanism – virtqueue. 25
3.3 (a) Scheme of assigning I/O devices directly to a virtual instance. (b) The difference between direct assignment and hardware SR-IOV solution. 26
3.4 SR-IOV scheme. 27
3.5 Goals of the scalable I/O virtualization. 28
3.6 (a) Direct assigning and sharing I/O devices using Intel® VT-d (IOMMU) with the SR-IOV device support. (b) Scalable I/O virtualization using replication and virtual device composition, diagram shows different data and control path. 28
3.7 A basic device assignment use case in KVM/QEMU. 29

3.8 A basic device assignment use case in KVM/QEMU without VFIO. 30
3.9 VFIO-mdev software framework. 31

4.1 The difference between a pure virtio and the vhost-net architecture. 33
4.2 Detailed schematic design of the new `npp_vhost` driver over the NDK platform for I/O virtualization. 34
4.3 Detailed schematic design of the new `vfio-mdev` driver over the NDK platform for I/O virtualization. 35

5.1 Schematic of the `mtty` driver functions and their interaction. 40
5.2 Schematic of the `vfio_iommu_type1`, the relationship between the structures, functions and their interactions. 45
5.3 The composition of the `nfb_mdev` driver, showing the interactions, operations and functions. 50

D.1 Instance of a virtual machine setup using a virtual-manager graphical UI. . 74
D.2 Configuration of a mediated device in XML format inside a software virtual-manager UI. 75

Chapter 1

Introduction

Many IT professionals think of virtualization only in terms of virtual machines (VM) and their associated hypervisors and operating-system implementations, but this view only scratch the surface of virtualization. Nowadays an increasingly broad set of virtualization technologies are redefining major elements of IT in organizations everywhere. Examining the definition of virtualization in a broader context we define virtualization as the art and science of making the function of an object or resource simulated or emulated in software identical to that of the corresponding physically realized object. In other words, we use an abstraction to make software look and behave like hardware, with corresponding benefits in flexibility, cost, scalability, reliability, and often overall capability and performance in a broad range of applications. Virtual machines trace their roots back to a small number of mainframes from the 1960s. Since then it has become an important aspect established in the mainframe world, a big data centers these days. With the introduction of Intel's 386 in 1985, virtualization started to take a place in the microprocessors – at the heart of personal computers. Contemporary VMs, implemented in processors with the required hardware support and with the help of both hypervisors and implementations at the OS level, are essential for computing productivity everywhere, most importantly capturing machine cycles that would otherwise be lost in today's highly capable 3+ GHz processors. VMs as technology could also provide additional protection, integrity and convenience, with very little overhead computing.

The goal of this project is to explore and find information about specific technologies associated with virtualization of network I/O devices and use this knowledge to select an appropriate technology to design and develop new kernel software drivers. The drivers should be compatible with the Linux operating system and should be used for the I/O virtualization of the hardware-accelerated FPGA network cards developed by the Liberouter[1] research group. Liberouter group is part of the CESNET association and the primary aim of the research is focused on hardware acceleration of network security and monitoring tools using FPGA cards.

The thesis is divided into multiple chapters. Chapter 2, features a theoretical analysis of the concepts and technologies associated with this project. This chapter also describes technologies like KVM virtualization, PCI-Express bus, IOMMU and NDK platform for the development of accelerated network cards but also discusses the question: *Why virtualize?* – the strengths and weaknesses arising from the use of virtualization. Chapter 3 is devoted to more detailed information regarding I/O virtualization; here are mentioned virtualization

[1] https://www.liberouter.org

technologies (divided into I., II. and III. generation) from software I/O virtualization to scalable hardware virtualization of I/O devices. Chapter 4 provides a design proposal for the problem based on this project's analysis and aims. The implementation process and issues connected with it are described in chapter 5. Chapter 6 provides detailed evaluation and a testing scenario of the drivers. Last chapter 7 summarizes achieved work with insight for future improvements.

Chapter 2

Concepts and technologies associated with virtualization

This chapter contains theoretical knowledge to form a basis for the design and implementation part of the thesis. First, it introduces and discusses virtualization itself, the benefits and drawbacks. Section 2.2, is about the essence of virtualization in the Linux operating system. Section 2.3 provides a software engineer's analysis of the PCI-Express bus as this bus is used by the FPGA accelerated network cards that we will use for virtualization. For I/O virtualization, it is usually necessary to collaborate with the host system's hardware (outside the I/O device itself). A representative of this group (inside the physical system) that matters the most is an IOMMU unit for the correct translation of virtual addresses to the physical – section 2.4. Last section 2.5 summarizes the information about the NDK platform over which the FPGA network cards are being developed.

2.1 The art of virtualization

Virtualization refers to the process of running a virtual machine (abbr. *VM*, also *virtual instance*), that acts like a real computer system. The difference between a real machine and a virtual instance is that it runs on a separate layer from the underlying hardware resources. Commonly, virtualization involves running multiple operating systems simultaneously on the same physical computer system. Virtualization itself hides users (*guests*) from the physical characteristics of the computing platform (known as *host*, *VMM - Virtual Machine Monitor*) and instead represents an abstract computing platform. Applications running on top of a virtualized computer may appear to be on their own dedicated computer, they don't know they are virtualized. Operating system, libraries and other programs are unique to the guest virtualized system and may not be directly connected to a host system that sits beneath it. A dedicated virtualization manager application (called a *hypervisor*) must be running on the host to manage each virtual instance. For more details about virtualization principles, see resources [4, 27, 18, 28]. Figure 2.1 shows the basic concept of virtualization described above.

Each technology has its **advantages** and **disadvantages**, virtualization is no exception:

+ The most beneficial aspect of virtualization is the more efficient usage of hardware resources, which leads to reduced operating costs. Virtualization allows us to use one host computer to run multiple virtual machines with different services. It is estimated that IT companies have an annual hardware cost (also with external service

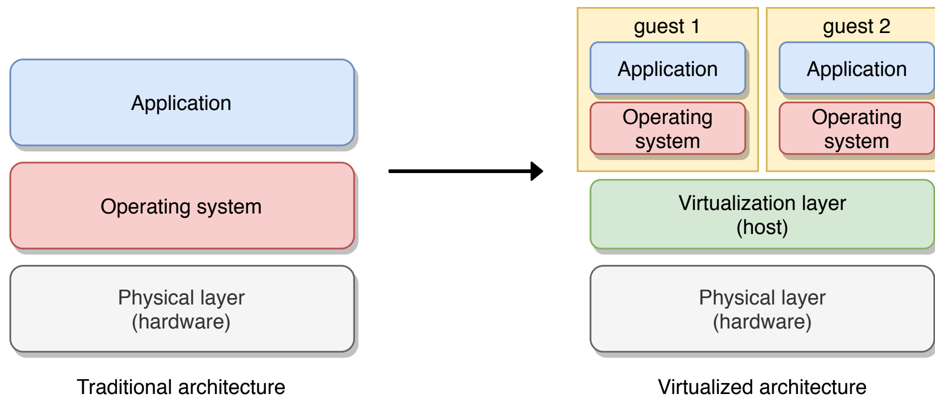


Figure 2.1: Scheme of common computing architecture compared to a virtualized computing architecture that features an abstract virtualization layer.

provision) ranging from around 35% to 50% of the total company budget[1]. If a service is operated only on one separate physical host, the service uses approximately 15% to 50% resources of the modern server (hardware is still more and more powerful each year)[2]. This way we lose the remaining resources that can be utilized for another task (in question is of course also factors such as consumption, scalability of the whole architecture, etc.). It means that in the long run, the manageability and operating cost is unnecessarily increased and therefore it is necessary to reflect and think about the usage of virtualization.

+ Increasing the number of physical servers is expensive and time-consuming. The reasons behind it are activities such as requirements for a new physical space, new hardware purchases, cabling server installation, etc. Once the virtualization is used correctly, the entire operation can be quicker and easier. No new hardware expenses, you only need set up your virtual instances properly. Virtualization seems to be the perfect flexible technology.

+ Virtualization brings the benefit of complete isolation of VMs, which is a good use case to test new environments with completely different configurations and operating systems as a workbench. Creating a new virtual instance is extremely simple and in some cases we can also emulate different architectures from our physical hardware.

– Running multiple services on the same server raises security issues. If many physically demanding services are operated on one physical machine, this may cause a significant slowdown or failure of the available services (the hardware could be overloaded, for example if a *Distributed Denial-of-Service* attack is executed against one service, it will affect other services). Available services could be very unreliable, thus paralyzing the existing virtual servers. Another case of security-related disadvantage is imperfect isolation (one of the core virtualization conditions) of each instance. Running virtual instances can overwrite private data, which should be in principle accessible only to a particular instance (this property is mainly reflected in I/O virtualization).

[1] http://omtco.eu/references/sam/it-costs-the-costs-growth-and-financial-risk-of-software-assets/

[2] https://confluence.atlassian.com/doc/server-hardware-requirements-guide-30736403.html

– The fact that new hardware technologies bring new technical concepts, solutions and methods makes it harder for virtualization to be easily developed. To make virtual environment fully compatible with hardware requires new software (drivers), which is not an easy task. There is therefore a great effort to support virtualization in each of the IT industry segments, especially by developing software solutions - drivers.

– Initial purchase price for buying new compatible virtual servers is also drawback. The price is almost always greater than the price of a similar physical server which doesn't support virtualization. Deploying virtualization often requires powerful hardware, the latest state-of-the-art hardware-enabled technologies, virtual instance management software, and various licenses that the customer has to pay for.

Virtualization of I/O operations

Computer systems are commonly used to perform some I/O operations: copying files between media (e.g., hard disk), playing video (graphics card), networking and transferring data (network card), sound (audio card), etc., which means the system contains multiple I/O resources (physical resources). When we use virtualization, these resources are shared among virtual instances.

A basic rule is that virtual instances must not interfere with each other (otherwise the concept of security/isolation would be violated). Also VM must be somehow isolated from actual physical hardware, or there must be an enforced policy that shares hardware based on exclusive ownership to each VM (i.e. each resource can be assigned and used only by one instance). For example, CPU cores (in multi-core processors) can be shared in the same way as the system scheduler shares cores between running processes in a modern multi-tasking operating system – each virtual instance has an assigned CPU core for one given time slot.

On the contrary, RAM (*Random–Access Memory*) can be shared exclusively by assigning a portion of physical RAM to each virtual instance. The memory management unit (*MMU*) must provide mapping (translation) from the virtual address space of VM to the physical addresses of hardware. In external memory (I/O devices), the MMU concept is extended to IOMMU (*Input—Output Memory Management Unit*), which is mainly used for correct direct memory access (*DMA*). More detailed information about IOMMU can be found in section 2.4.

Hardware sharing brings a problem because physical hardware is usually designed to be used only by one OS. In some cases, sharing devices without breaking the isolation is simply impossible. If e.g., hard disk is shared, the write operations of each VM may affect the state of other virtual instances. Similar problems arise also when sharing network cards. Network cards typically contain specific host settings – hard-coded physical address of the adapter, switching logic for delivering data to chosen ports, etc. More on ways how to solve I/O hardware virtualization is in the chapter 3.

2.2 Virtualization basics in the Linux operating system

There are several implementations of different virtualization levels under the Linux operating system: Project KVM (*Kernel-based Virtual Machine*), VirtualBox, Wine emulator, Xen, VMware or lightweight virtual machines such as Docker container and others. The subsequent text is focused only on the KVM project for the following reasons: Wine serves as a pure emulator of software architecture and therefore does not fall under the I/O vir-

tualization, Docker virtualizes at the application level of the operating system – we cannot virtualize I/O, VMware is closed commercial architecture; similarly VirtualBox is closed proprietary software and mentioned Xen project is open-source, but it takes more effort to incorporate working drivers into the final architecture, some parts of the Xen subsystem are partially commercial.

The virtualization and emulation architecture of KVM/QEMU

There are two types of virtualization hypervisors (*virtual instance managers*). Type 1 hypervisor (*bare-metal hypervisor*) is in direct control of all resources of the physical computer. It runs without any necessary additional software connected directly to the hardware – it also uses less resources of the physical system, but it must have compatible drivers for all hardware (more efficient but difficult to solve). In contrast, a type 2 hypervisor (*hosted hypervisor*) operates either "as part of" or "on top of" an existing host operating system. It is based on a full-fledged operating system – hypervisor uses the device drivers of the operating system in which it runs.

The KVM (*Kernel-based Virtual Machine*) project is an open-source virtualization architecture for Linux distributions. KVM turns the Linux kernel into a type 1 hypervisor and KVM has both host and guest support in the upstream Linux kernel since 2007[3]. The Intel VT-x or AMD-V virtualization hardware extensions on the host platform need to be available for running KVM module. When converting the kernel host system to a hypervisor, KVM can take advantage of already implemented components inside kernel, instead of re-implementing them from scratch. Thus, KVM uses system scheduler, memory manager, device drivers and more. From a host prespective, any created instance is treated as a standard Linux process.

KVM is often used in combination with QEMU (*Quick EMUlator*) software that runs in the user space. QEMU is an open-source software virtualization emulator – emulates a computer's processor through dynamic binary translation and provides a hosted instance of various hardware and device models. This makes QEMU a type 2 hypervisor. QEMU has a long list of peripheral emulators including disk, network, VGA, PCI, USB, serial ports, parallel ports, etc. Emulation allows QEMU to run different guest operating systems (different architectures such as ARM, x86-64 and others). QEMU is typically deployed with KVM to run virtual instances with close-to-native hardware speeds (using the benefits of hardware extensions such as Intel VT-x or AMD-V). Figure 2.2 illustrates the Linux architecture of the host system in a hierarchical representation, featuring a kernel-level KVM module and a user-space QEMU module in VM virtualization.

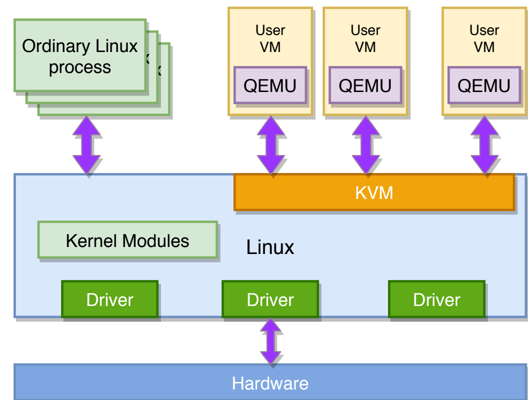


Figure 2.2: Linux based host system architecture virtualized with KVM/QEMU.[4]

[3] https://www.kernel.org
[4] Source: https://medium.com/@jain.sm/kvm-and-qemu-as-linux-hypervisor-18271376449

Application programming interface libvirt

In addition to creating a virtual environment itself, VM also needs to be managed. Libvirt is a collection of open-source software consisting of an application programming interface (*libvirt API*), daemon (*libvirtd*) and virtualization platform management tool (*virsh* - command line utility). This collection of software provides a convenient way to manage virtual machines and other virtualization features such as data storage, network interface management, and more. The primary goal of libvirt is to provide a unified way of managing multiple different virtualization hypervisors. It can be used to manage KVM/QEMU, Xen, VMware ESXi and other virtualization technologies.

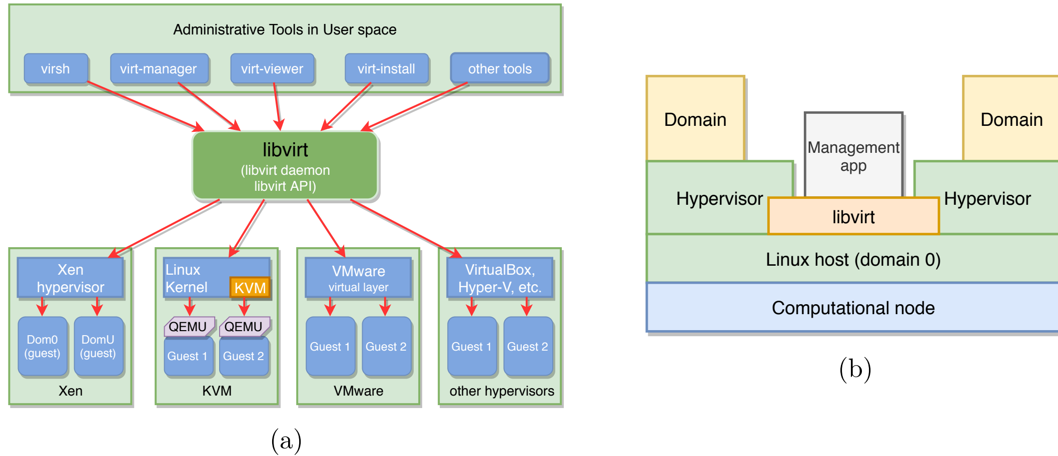


Figure 2.3: (a) Hierarchical details of libvirt architecture.[5] (b) Level view of libvirt implemented in a virtualized environment.[5]

2.3 PCI-Express bus from a software engineer's point of view

PCI-Express (abbr. *PCIe*, full-text. *Peripheral Component Interconnect Express*) is a high-speed serial bus designed for computer systems. PCIe replaces older standards, in particular outdated PCI (parallel bus) and AGP (parallel bus). PCIe extends and re-implements parts of PCI. The PCIe bus holds title of the most widespread bus in today's modern systems. It's purpose is to interconnect devices inside the computer such as graphics cards, hard drives, network cards, etc. The PCIe standard specification is maintained and developed by PCI-SIG (*PCI Special Interest Group*), a group of over 900 companies from the industry. PCIe exists in various variations of link width, as is shown in the table 2.1. PCIe standard also exists in multiple versions (each new version is gradually developed). Newer versions improve data throughput (by making changes in coding, physical specification changes, etc.) and add new options to the bus itself, for example enabling DMA, virtualization and other technologies. The table 2.2 shows throughput of different versions of the PCIe bus.

Link width	x1	x2	x4	x8	x16	x32
Throughput in GB/s	0.5	1	2	4	8	16

Table 2.1: Display of throughput (one-way) to the number of lines in PCIe 2.0.

[5]Source: http://yfchang.blogspot.com/2013/08/virtulization-libvirtkvmqemu.html

Version	Throughput per line	Total "theoretical" throughput x16
PCIe 1.x (2003)	2 GB/s	8 GB/s
PCIe 2.x (2007)	4 GB/s	16 GB/s
PCIe 3.x (2010)	8 GB/s	∼32 GB/s
PCIe 4.0 (2011)	16 GB/s	∼64 GB/s
PCIe 5.0 (2019)	32 GB/s	∼128 GB/s

Table 2.2: Different versions of PCIe and their throughput.

PCI addressing

Because PCIe is an extension of PCI, there will be no distinction between PCI/PCIe in the following text. From the software developer standpoint, PCI supports device auto-detection, the PCI devices are automatically configured during a boot phase of the machine. Therefore the device driver must have access to the configuration information to complete the initialization. Each PCI peripheral device inside a system is identified by the *bus number*, *device number* and *function number*. The PCI specification specifies up to 256 available buses that one system can host. Unfortunately 256 buses are not enough for larger systems, especially systems in computing clusters, hence the Linux operating system supports the so-called PCI domain concept. There could be a multiple domains in system, the PCI domain can contain up to 256 buses, each bus can contain a maximum of 32 devices, and each device can perform an activity with a maximum of 8 functions. This provides plenty of room for the addressing space. Modern systems contain at least two PCI buses. For seamless coexistence of more buses than one in a system, a PCI bridge needs to be involved – special peripheral PCI device whose task is to connect two buses internally. The overall arrangement of the PCI subsystem is therefore a tree hierarchy with a root (host bridge, PCI bus 0). Each bus is connected to a higher layer bus, up to bus 0 (root). Figure 2.4 illustrates the typical PCI subsystem with highlighted PCI bridges.

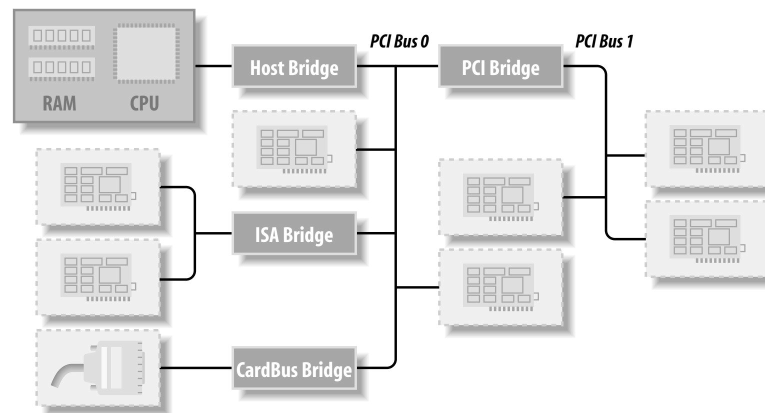


Figure 2.4: Scheme of a typical PCI subsystem including the devices and their connection to the buses and PCI bridges. Source [5].

PCI peripherals are addressed via 16-bit hardware addresses. If a Linux PCI domain is added, a 32-bit address is created, divided into a domain (16 bit), a bus (8 bit), a device

(5 bit) and a function (3 bit) address. As an example of device addresses (disk controller, network adapter) in some system, see the following listing:

```
$ lspci | cut -d: -f1-3
0000:00:12.0 Ethernet controller: Intel Corporation I350 Gigabit Network
Connection
0000:00:1f.2 SATA controller: Intel Corporation C600/X79 series SATA AHCI
Controller
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
|-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
```

Listing 2.1: Sample addresses of PCI devices in the system.

PCI hardware devices support three different types of memory spaces: configuration space (configuration registers), default memory space and I/O space (ports). The memory and I/O spaces are shared within all devices on the same PCI bus. The configuration space is unique and not shared, only one slot may be addressed at a time. The configuration space contains key information about each PCI device. The space is divided into special data structures with a size of 256 bytes. There are two types of configuration space – type 0 designed for endpoints and type 1 for bridges (switches). Furthermore, the memory space differentiates itself between either prefetchable memory — memory that can be pre-read into the cache and non-prefetchable memory — memory must be accessed directly. Configuration transaction mechanism performed by kernel manages access configuration registers of a PCI device. Memory and I/O space are usually accessed via kernel functions `ininb()`, `readb()`, `readw()`, etc. The I/O space in the PCI bus uses the 32-bit address (this limits to 4 GB of I/O ports), while the memory space can be accessed using 32-bit/64-bit addresses. Figure 2.5 explains memory layout of the PCI.

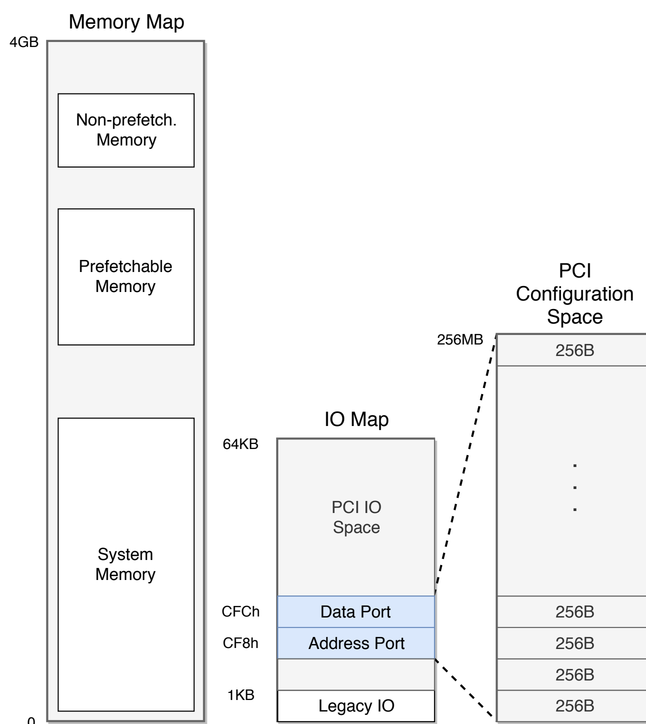


Figure 2.5: PCI memory layout preview.

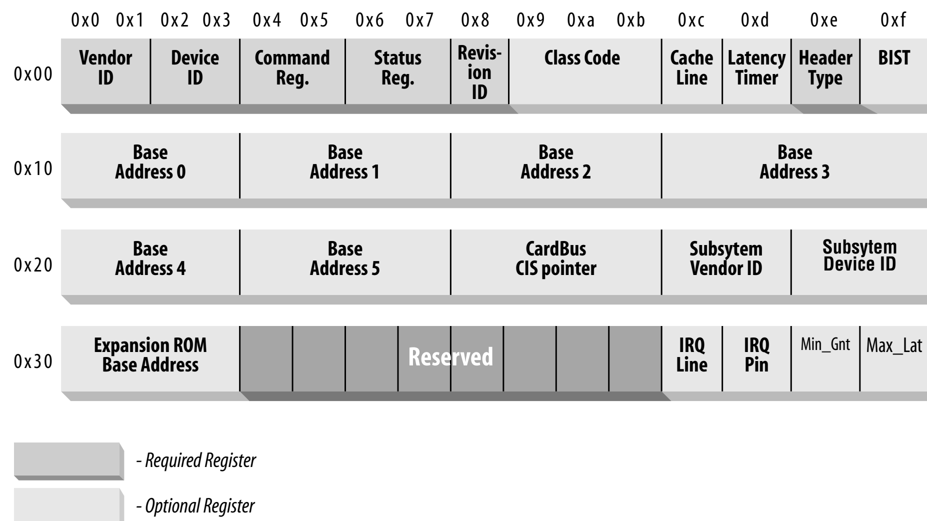


Figure 2.6: The PCI configuration space. Source [5].

PCI configuration space/registers

All PCI devices have at least 256 byte address space. The first 64 bytes are standardized by the PCI specification, while the rest are device-dependent. Figure 2.6 displays the layout of the device-independent configuration space. As shown in the figure, some configuration registers are required and others are optional. Each PCI device must contain valid values in the required registers, while the contents of the optional registers are not used unless the contents of the required fields imply that they are valid. The values in the PCI registers are written in the form of little-endian, the lowest value of the byte (LSB - least significant bit) is at the lowest address. The other bytes follow in ascending order.

To describe all mandatory register values is beyond the scope of this text, their exhaustive description is in the official PCI specification[6]. Three compulsory registers are used to identify device, the `Vendor ID`, `Device ID` and `Class Code`. These registers are read-only, and each PCI device manufacturer assigns the correct values to these registers. The values in the `Subsystem Vendor ID` and `Subsystem Device ID` registers can be set up and help to distinguish similar devices from one manufacturer.

`Vendor ID`	the manufacturer's identification number, assigned by the central authority PCI-SIG
`Device ID`	PCI device identification, selected by the manufacturer
`Revision ID`	revision number, selected by the manufacturer
`Class Code`	identifies the generic functionality, e.g., multimedia, network equipment, etc.
`Subsystem Vendor ID` `Subsystem Device ID`	identification of the possible subsystem within a given PCI device

Table 2.3: Description of the registers identifying the PCI device.

[6] https://pcisig.com/specifications

Bits	Description	Values
For all PCI BARs		
0	Region type	0 = Memory 1 = I/O (obsolete)
For Memory BARs		
2-1	Location	0 = any 32-bit 1 = <1 MiB 2 = any 64-bit
3	Prefetchable	0 = no 1 = yes
31-4	Base Address	**16-byte aligned**
For I/O BARs (obsolete)		
1	Reserved	
31-2	Base Address	4-byte aligned

Table 2.4: Detailed description of individual bits in the PCI Base Address Register (BAR) used for memory allocation.

Process of PCI device memory allocation

After the driver has identified the device, it usually needs to read or write to the three address spaces mentioned above. Accessing the configuration space is vital to the driver because it is the only way the driver can find information about PCI mapped memory spaces in the system. Memory allocation (after system boot) uses base address registers (BARs) – the base address of the the device. In total, up to 6 blocks of memory can be allocated (6 BARs). If a 64-bit addressing is enabled, two registers are used instead of one (`BAR0` together with `BAR1` form a pair, `BAR2` with `BAR3`, etc.). After powering-on or restarting the device, the contents of the BAR registers are in an uninitialized state (table 2.4). The lower bits of the BAR registers are hard-coded by the manufacturer and these bits determine what size and type of memory a device requires for its functionality.

The steps of procedure for handling a PCI memory allocation request are as follows:

1. The driver fills the uninitialized BAR register with all ones. The bits that are set by the manufacturer will not be affected (they are hard-coded).
2. The driver reads back the value of the BAR register and detects the amount of required memory and type of memory. Based on this, the system/driver allocates required memory space (system will generate base address).
3. The driver writes a corresponding base address value of allocated memory into the BAR register.

PCIe capabilities

In addition to the described configuration space in the previous section, the configuration space also contains various additional information about different capabilities of the PCI peripherals. The first 192 bytes of the PCI configuration space define a capabilities list, to allow more parts of configuration space to be standardized without conflicting with existing uses. Each capability has an eight-bit value (one byte) that identifies the type and format of a PCI-Compatible Capability structure (`capability ID`), and one byte to point to the next capability (`next capability pointer`, capabilities list is in principle a classical list

data structure). The number of additional bytes depends on the `capability ID` (each capability is different). If capabilities are used, a bit in the *status register* is set, and a pointer to the first of the capabilities in a linked list is provided in the `capabilities pointer` register defined in the standardized PCI registers. The example of a whole PCI configuration space with its capabilities core is illustrated in figure 2.7. These PCI/PCIe capabilities for example could be:

- MSI support (Message Signaled Interrupts) replacing default interrupts using chained signals.
- DMA transaction support – device contains a controller that supports direct memory access.
- Vendor Specific Extended Capability (*VSEC*), used by manufacturers to allow them to specify their own extensions. To give an example, NDK platform is using VSEC to specify where to find the whole hardware description of a given card in a Device Tree format (the usage of Device Tree are in the author's previous work [26]).
- Support of PCIe device virtualization (e.g., a SR-IOV technology). The SR-IOV enabled device is defined to have at least one physical function (*PF*) and multiple virtual functions (*VF*). PF is the standard PCIe function[7]. It has a configuration space and the host software manages it just like any other PCIe function (PF has the addressable space, see 2.3). Additionally, it supports standard operations such as enabling/disabling the host device, power management and so on, but it also provides functionality to create - allocate, remove or configure new VFs. The VF is a lightweight PCIe function that implements only a subset of the standard PCIe feature components. For example, it does not have its own power management, it shares capabilities with its superior (parent) PF and the VF itself cannot allocate, remove or create additional VFs. PF is superior to multiple VFs. SR-IOV details, chapter 3.2.

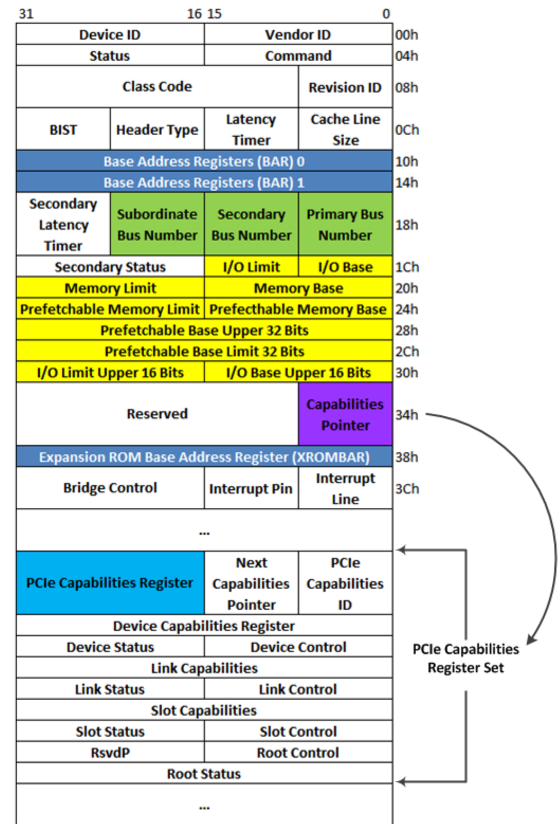


Figure 2.7: An example of a whole PCI configuration space with its capabilities.

- PASID (*Process Address Space ID*) Extended Capability Structure. The presence of a PASID Capability indicates that the endpoint supports sending and receiving PCI TLPs (*transaction layer packet*) containing a PASID TLP Prefix. PASID is a unique

[7] with or without virtualization each PCI device has always exactly one PF

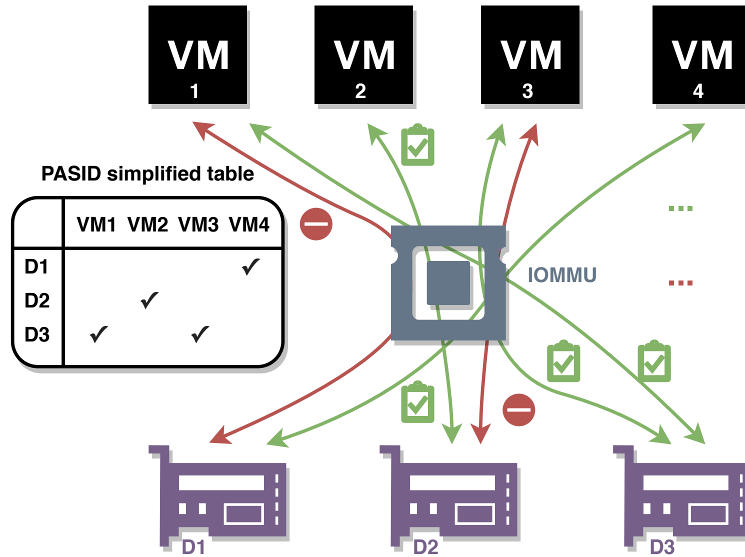


Figure 2.8: An simplified example of PASID concept.

ID associated with a process (in our case, a guest VM) and thus with a page table for memory translation this leads to better device isolation. Each transaction within a PCI bus is checked by IOMMU and the device can tell IOMMU which process they are interested in. If PASID bind requirement between process and device is not met, IOMMU will simply deny (discard) ongoing communication. For a better understanding of this concept see figure 2.8 which shows simplified example of 3 devices and 4 processes (VMs) with a PASID table that binds devices with VMs. For example if process VM1 tries to communicate with the first device (or vice versa), communication will be unsuccessful. The PASID is a cornerstone for proper design of a working PCI virtualization within IOMMU. More detailed information about PCI PASID can be found in PCI specification [24], exclusively in chapters 6.20 and 7.8.8. Structure of a PASID capability itself is shown in figure 2.9.

- Intel® Scalable I/O virtualization capability inside PCI Express Designated Vendor Specific Extended Capability (*DVSEC*). This capability is defined for systems software and tools that may need to detect endpoint devices supporting Intel Scalable IOV, without host driver dependency. More about it in [11].

More about the PCI-Express bus and its components can be found in sources [20, 31, 5, 23, 4, 24, 11].

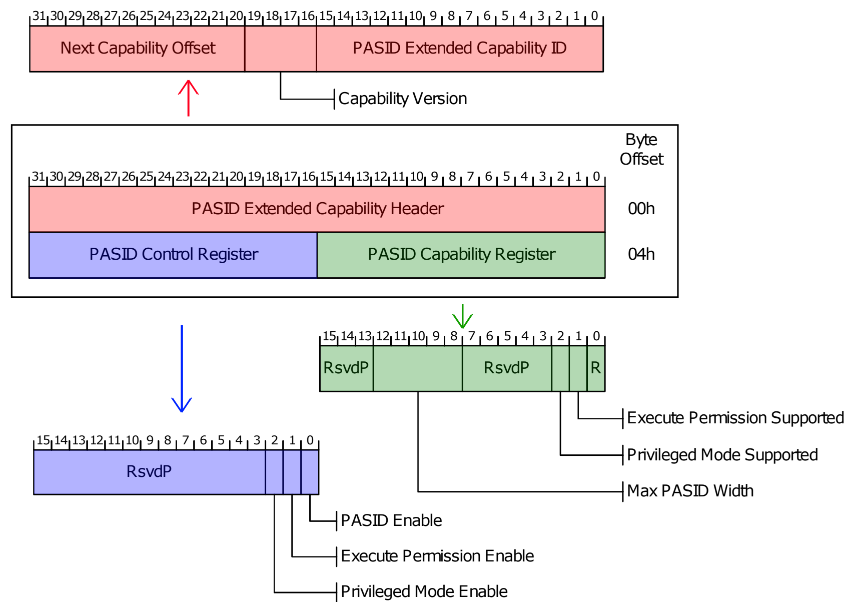


Figure 2.9: A composition of a PCI PASID Extended Capability Structure.

2.4 IOMMU and direct memory access

The IOMMU (*Input–Output Memory Management Unit*) is a memory management auxiliary unit that interconnects an operating memory with an I/O bus and carries direct memory access (DMA) address translations. Similar to the traditional MMU (*Memory Management Unit*) that translates – virtual processor addresses to physical hardware addresses, the IOMMU maps the virtual addresses visible on the device (I/O addresses) to the physical addresses. This unit is part of the motherboard chipset or as a separate component of the processor. Some IOMMU units (depends on CPU architecture) also provide memory protection against faulty or malicious devices. The MMU and IOMMU within the system can be seen in figure 2.10.

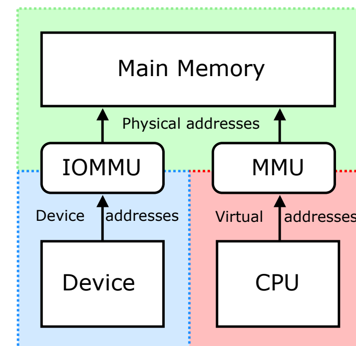
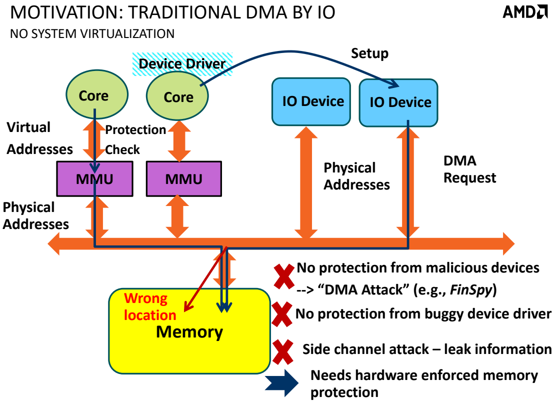


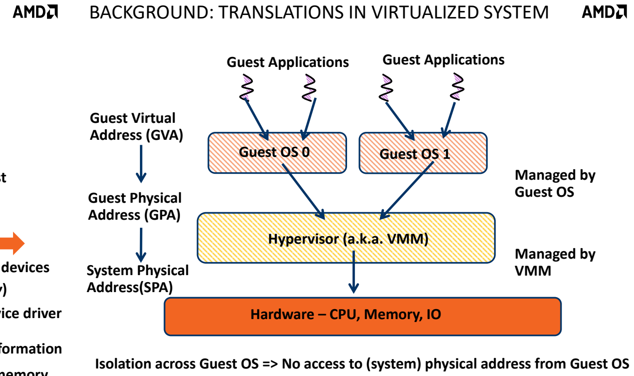
Figure 2.10: A schematic representation of MMU and IOMMU memory management units.[8]

If the target system does not contain an IOMMU unit, it presents a number of problems that need to be tackled. The DMA transfers between the processor and the I/O device without virtualization - figure 2.11a, without a IOMMU unit, raise a problem of invalid location. Processor cores and I/O devices may have invalid shared addresses between them – an example of a partial solution could be fixed addresses (an unacceptable long-term solution, leading to poor development practices). Another problem is zero protection of memory regions from malicious devices or broken devices connected to the bus. There are also threats of a variety of side channel attacks, all of this is happening because there is no controller in the system to prevent this. If virtualization is used, translation of the guest virtual addresses (GVA) $\rightarrow$ to guest physical addresses (GPA) $\rightarrow$ and those to system physical addresses (SPA, hardware addresses) is

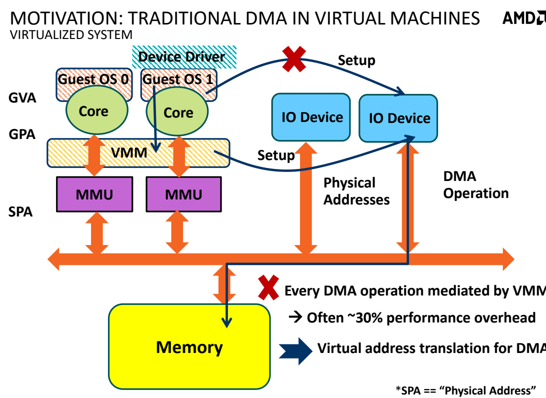
[8]Source: https://en.wikipedia.org/wiki/Input-output_memory_management_unit



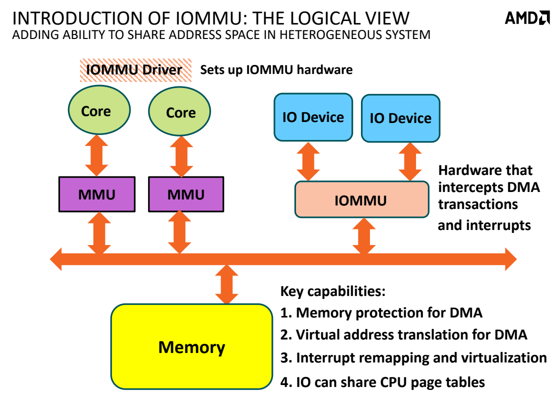
(a) DMA transfers in a classic system without virtualization.



(b) Multi-level translation of virtualized addresses.



(c) DMA transfers in a virtualized environment.



(d) Introduction of the IOMMU into a system.

Figure 2.11: DMA concepts w/o virtualization, w/ virtualization and why it is necessary to introduce an IOMMU unit. Source [16].

required. This adds new levels of translation - figure 2.11b. There are currently several ways to speed up direct translation of virtual addresses to physical ones.

Different specialized page tables (each architecture = individual approach) are used, distinct with its own pros and cons: Oracle - *shadow page tables*, Intel - *extended page tables* or AMD - *nested page tables*. More about paging in a virtualized environment can be found in [22]. If we look again at DMA transfers, this time in a virtualized environment but still without IOMMU - figure 2.11c, each direct memory access must be mediated through the host-based hypervisor, which in practice means performance loss. For the reasons mentioned above it is necessary to introduce a new hardware unit for memory management into the system – already mentioned IOMMU, case shown in figure 2.11d. Deploying an IOMMU unit into a system provides at least 4 key features:

1. Memory protection for DMA transfers within I/O peripherals.

2. Support the virtual addresses translation for DMA transfers when using a virtualized environment.

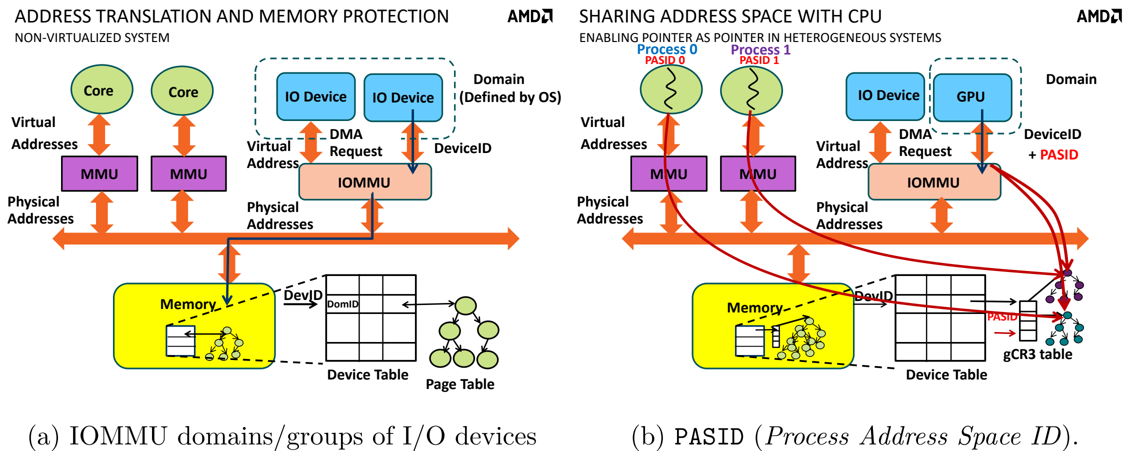
3. Better I/O device interrupt management in a virtualized environment.

4. Under certain conditions, support sharing memory pages between the processor and the I/O device.

In addition, the IOMMU can take advantage of virtual addressing by being able to use large contiguous virtual areas, even if the physical memory is fragmented. The IOMMU can also use parts of the operating memory that cannot be directly physically addressed (e.g., a 32-bit x86 platform with more than 4GB of memory can be addressed using Physical Address Extension (*PAE*), but the normal 32-bit PCI device cannot be addressed above this 4GB limit). Devices can address the entire memory via the IOMMU, avoiding the time overhead associated with copying buffers into the peripheral's address memory space.

IOMMU groups, domains and `PASID` interoperation

Memory protection works in IOMMU as well. This protection is against software or hardware faults in I/O devices or drivers (system recovery when writing to incorrectly addressed memory) and against potentially malicious attacks against software or I/O device drivers. Building trust for each device depends on the operating system. For this purpose serves a so called IOMMU groups/domains concept[9], illustrated in the 2.12a figure. If multiple devices are assigned to a single domain, they can communicate with each other, share their memory spaces, etc. Addressing individual devices is by default based on `Device ID` from PCI (see 2.3). Each device is in some group/domain and the IOMMU verifies the rights when requesting access occurs. If the device does not have sufficient permissions, the IOMMU cancels the request, protecting the memory.



(a) IOMMU domains/groups of I/O devices

(b) `PASID` (*Process Address Space ID*).

Figure 2.12: Concepts of an IOMMU addressing within a domain/group, device and process (by `PASID`). Source [16].

In an operating system, each process has its own address space (virtual address space). A conceptual problem arises, what if two different independent processes want to communicate with the same I/O peripheral address space – the device addresses must be mapped into the shared address space. The IOMMU unit will help by keeping track (translation) of these address spaces. To distinguish between the address spaces of individual processes,

[9]There is a slight difference between an IOMMU group and an IOMMU domain. Difference is in a various level of granularity and control-ability. The groups are created at a boot time by an IOMMU driver, each device is in a separate group, domains are managed by the operating system or alternatively by a user. Normally each group is in a one domain, mapping is 1 to 1.

PASID is used (figure 2.12b) for identifying the necessary page tables (cached in IOMMU) for successful translation. IOMMU is a relatively new technology and various improvements and technical specifications are constantly appearing, especially in the software development – support for operating systems and device drivers. The use of IOMMU has some disadvantages, it reduces performance due to overhead page translation/management costs and also the need for larger cache sizes for added I/O pages. More about IOMMU can be found in [4, 16, 22].

2.5 Netcope Development Kit (NDK) platform

Netcope Development Kit (abbr. *NDK*) platform is a framework (consisting of tools, software, firmware, intellectual property cores[10], hardware, etc.) designed for the rapid development of hardware-accelerated network applications based on FPGA chip technology. NDK offers a comprehensive environment enabling fast prototyping of various applications in the shortest possible time. NDK components are designed to achieve the maximum throughput when processing network data transfers at speeds up to 100 Gb/s and more. The hardware card is connected to the host system using a PCI-Express bus.

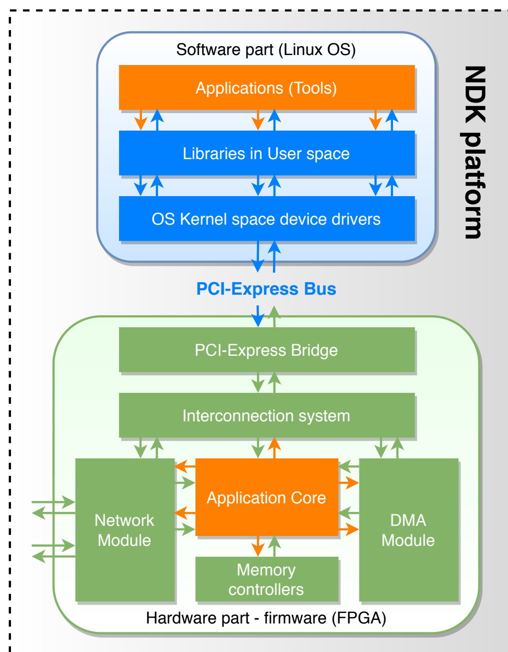


Figure 2.13: NDK Platform for the development of hardware-accelerated network applications over FPGA technology.[11]

NDK device drivers for the Linux operating system allow the user transparent specific functions, such as PCI-Express bifurcations[12], work with standard network interfaces in Linux, etc. PCI transactions are used for fast packet transfers between the FPGA and the software application. These transactions are highly optimized, using direct memory access without unnecessary copies of data (*zero copy*), which saves a lot of CPU cycles and memory bandwith. Scheme 2.13 represents important components of the NDK platform, consisting of the software parts (tools, libraries and drivers) and the firmware/hardware part (various application cores – firmware projects: e.g., NIC, HANIC, SDM, and others; more about application cores in [26]). Firmware projects for FPGA are programmed using the VHDL hardware programming language.

Figure 2.14 illustrates how the actual product built on the NDK platform looks like – specifically the NFB (*Netcope FPGA Boards*)/COMBO FPGA cards. The cards are prepared to be deployed to a server. For clarification of the figure, the cards have a massive passive coolers attached to them, two input ports for the optical network modules on the side and the PCI-Express interface on the bottom.

[10]https://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core
[11]Source: https://www.netcope.com/en/products/fpga-development-kit
[12]https://www.netcope.com/en/blog/100g-ethernet-data-stream

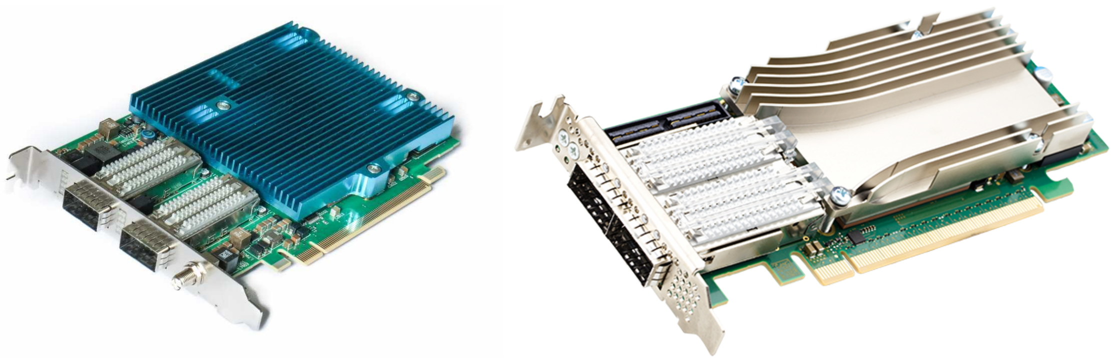


Figure 2.14: The photography of COMBO-100G2Q and NFB-200G2QL cards developed on the NDK platform.[13]

NDK software package

As mentioned before, the NDK platform is also composed of a software part. Right above the hardware layer begins the software layer (drivers), which is used to transfer data between the card and host system. Input/output buffer memory (ring buffers) is reserved for data transfers. The operating system driver has direct access to the address space of these buffers and provides a single program interface for the user tools that need to pass data from/to the card. The software part of an NDK platform is divided into 3 packages. Figure 2.15 illustrates the 3 different layers: drivers, library and tools. The `drivers` package creates kernel drivers to support NFB/COMBO cards, the main driver is `nfb` driver which creates internal nfb device structure to work with, listing 2.2. The interlayer between the applications and the card driver is formed by the `libnfb` library. The library provides the same interface to all applications. The rest – all the user tools for obtaining card operational parameters (listing 2.3), card management and configuration, data transfer between the card and system, are included in the `tools` package. Figure 2.16 shows a simplified hierarchical representation of the NDK platform's software part.

```
andre(SL7) ~$ lspci -vv
06:00.0 Ethernet controller: Netcope Technologies, a.s. NFB-100G1-e1
        Subsystem: Netcope Technologies, a.s. Device 0800
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- ParErr- Stepping- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- DEVSEL=fast >TAbort- <TAbort- <MAbort- >INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Interrupt: pin A~routed to IRQ 43
        NUMA node: 0
        Region 0: Memory at f0000000 (64-bit, non-prefetchable) [size=128M]
        Region 2: Memory at e8000000 (64-bit, non-prefetchable) [size=128M]
        Capabilities: <access denied>
        Kernel driver in use: nfb
        Kernel modules: nfb
andre(SL7) ~$ cat /dev/nfb0
```

Listing 2.2: Correctly initialized NFB-100G1 card in the system, using NFB drivers.

[13]Source: https://www.liberouter.org/technologies/cards/ and https://www.netcope.com/en/products/fpga-boards

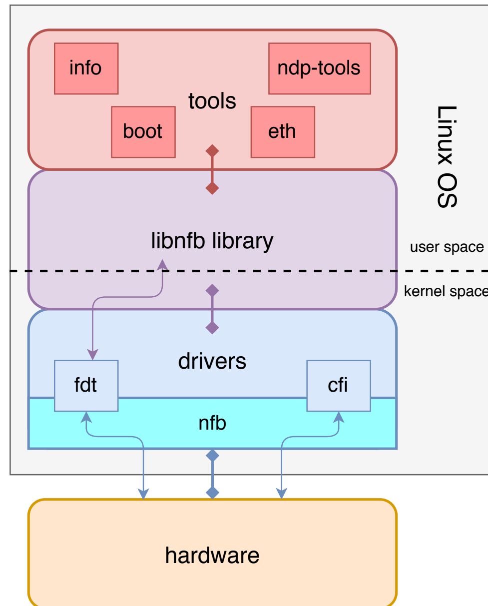


Figure 2.15: Software component scheme of NDK platform.

```
andre(SL7) ~$ nfb-info
--------------------------------------- Board info ----
Card name                   : NFB-100G
Serial number               : 1532
Network interfaces          : 1
------------------------------------ Firmware info ----
Built at                    : 2018-02-20 11:16:04
Build tool                  : Vivado v2017.4 (64-bit)
Build author                : spinler@cesnet.cz
RX queues                   : 2
TX queues                   : 2
ETH channels                : 1
-------------------------------------- System info ----
PCI slot                    : 0000:06:00.0
NUMA node                   : 0
```

Listing 2.3: Displaying the NFB-100G1 card information with the `nfb-info` tool.

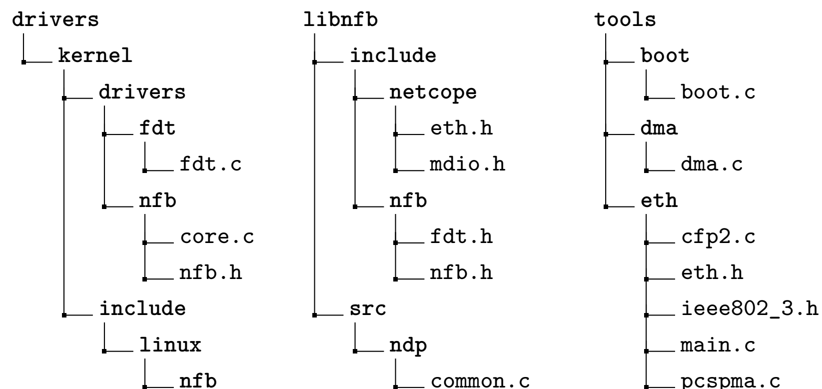


Figure 2.16: A hierarchical structure of the NDK platform software package including drivers, library and card tools.

Chapter 3

Technologies of network interface I/O virtualization

The I/O virtualization basics have been described in chapter 1. This chapter goes more into detail and discusses the different approaches to virtualizing network interfaces and their differences. The book [4] provides a broad overview of the various I/O virtualization techniques and about virtualized network interfaces (*NIC*) from a host and guest perspective. The goal of I/O virtualization technology is plain, to create a virtual network interface (*vNIC*) and assign it to a virtual instance. But to achieve this the vNIC interface needs to fulfill multiple conditions. First, the vNIC should provide good performance – high throughput and low latency (as close as possible to physical hardware NIC). Second, the vNIC needs to be flexible – the possibility of a simple migration of a virtual instance from one physical machine to another. This chapter is divided into three parts. The first part 3.1 describes virtualization without hardware support – software I/O virtualization, often referred to as I. generation I/O virtualization. The second part 3.2 is dedicated to hardware virtualization, namely the direct assignment of I/O devices to virtual instances. This solution often provides the best possible performance, but loses the flexibility and manageability of vNICs. Finally, the third part 3.3 describes the current trends in I/O virtualization of network interfaces – where scalability is the main motto. These technologies are often solutions on the edge between software and hardware I/O virtualization. The information presented in this chapter comes from resources [4, 28, 30, 9, 36, 11, 14, 6, 1, 35].

3.1 Software I/O virtualization – I. Generation

This group of virtualization technologies are mainly emulation and paravirtualization technologies. There is no need for special hardware support for the devices. The emulation works on the principle that the hypervisor (VMM in the host) selects some well-known hardware – for example an Intel PRO/1000 PCI network adapter. This adapter has supported drivers in many operating systems (e1000 driver in Windows, Linux and FreeBSD). The hypervisor subsequently creates a custom adapter, in the form of a fictional driver. The adapter will be attached to a virtual instance and the VM treats it as a fully working NIC (in fact, the VM sees it like a classic e1000 driver in form of a vNIC). The hypervisor emulates all communication between real hardware and a virtual instance in the software. The obvious advantage of emulation is the reliability and wide compatibility of most operating systems, but emulation overheads cause serious performance degradation and emulation is

therefore one of the slowest virtualization technologies. More advanced emulation can be achieved with hardware acceleration using various technologies – especially in the area of network adapters (see article [12]), unfortunately it's still not as effective as other presented virtualization concepts. A scheme 3.1a shows the I/O software virtualization, in the scheme you can see the fictional driver instances and the hypervisor emulation.

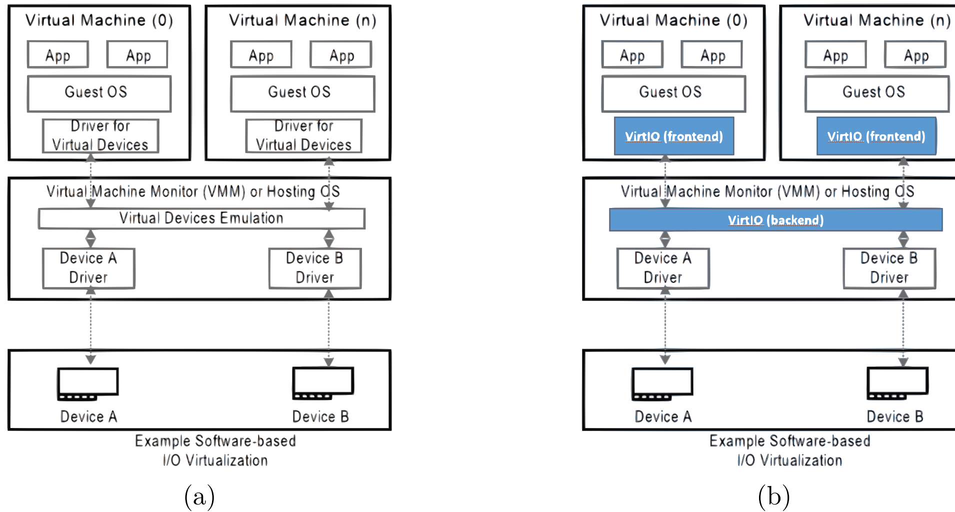


Figure 3.1: (a) Scheme of a software I/O virtualization. (b) Scheme of a software I/O virtualization using a virtio technology.

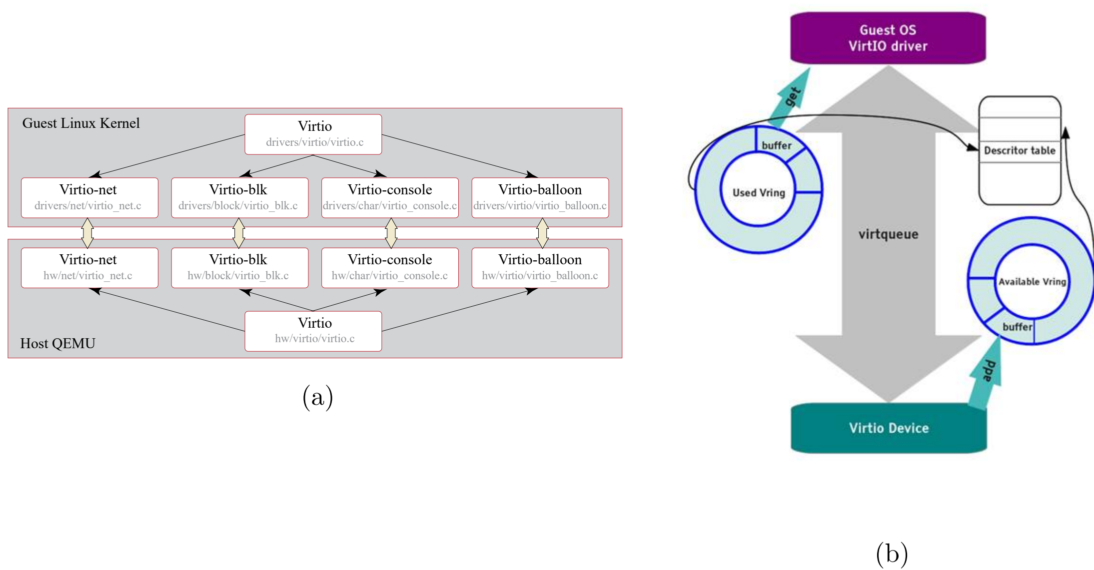
In general, the virtual instance must contain a fictional driver, and the hypervisor emulates the fictional driver – it converts communication into a real driver. There may not always be a uniform interface for communication between an instance and hypervisor. The consistency of the interface partially resolves the paravirtualization – e.g., using the standard *virtio* interface. If the virtual system more or less realizes that it is running in a virtualized environment, then it is a paravirtualization instead of an emulation. In particular, the paravirtualizing I/O device is similar to the emulation described above, except for the difference that a special virtual network card is emulated instead of a known hardware model. See difference between figures 3.1a and 3.1b. The paravirtualized drivers consist primarily of a front-end and a back-end part. The front-end driver is located exclusively inside the VM and the back-end driver in the host.

Transmission standard Virtio

Virtio is a standard describing the layer of paravirtualized devices in Linux. The standard has been published by the OASIS community [21], currently in version 1.1. The aim of the standard is to simplify the virtual device drivers and help to reduce the amount of source code in operating system drivers (so that only one standard interface is used). Virtio consists of several parts. The two main parts are: device drivers (mainly for front-end virtual instances) and single transport mechanism of data transfer between front-end and back-end drivers. The implementation of virtio is located in the Linux kernel, figure 3.2a shows the individual front-end implementation and their back-end device driver counterpart. Ring buffers (so-called *virtqueues*) are used as transport mechanisms for communication. Each virtqueue is composed of 3 elements:

1. Descriptor Area – used to describe buffers
2. Driver Area – data supplied by the driver to the device (known as Available Ring in the older version of virtio standard)
3. Device Area – data delivered by the device to the driver (known as Used Ring in the older version of virtio standard)

The described virtqueue elements are outlined in figure 3.2b. The device can have zero or more virtqueues – it depends on the type of device (network, disk, etc.) and on the interrupt system, how many interrupts the device can handle (virtqueue transmissions are signaled by the interrupts, or in the worst case it is possible to use CPU polling – active processor waiting, but that leads to a big performance loss).



(a)

(b)

Figure 3.2: (a) Virtio implementations in the Linux kernel. Source [4]. (b) Virtio transport mechanism – virtqueue.[1]

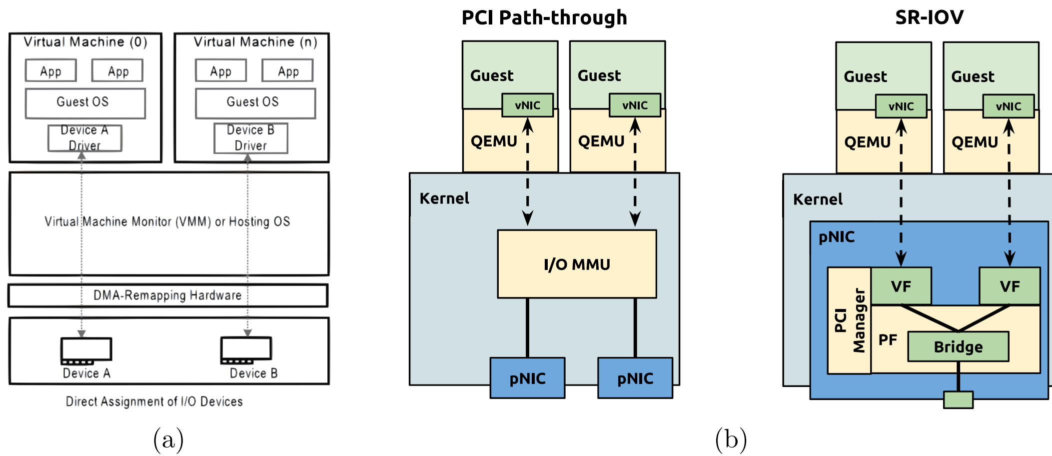
3.2 Direct I/O device assignment – II. Generation

Paravirtualization (emulation) creates software "fictional" network interfaces (see 3.1). However, in full virtualization (with hardware support), the actual hardware device is directly associated with the virtual instance. The virtual instance communicates with the hardware (physical NIC, *pNIC*) without additional hypervisor involvement. This is illustrated in figure 3.3a. Direct assignment of I/O devices poses a number of problems, but it offers the best performance (no need the data transfers between hypervisor and hardware) – with minor limitations, performance can be almost up to the level of the physical hardware itself.

Each virtual instance must contain the appropriate driver for the connected hardware – manufacturers need to create drivers for their hardware for different operating systems,

[1]Source: https://www.ibm.com/developerworks/cn/linux/1402_caobb_virtio/image002.jpg

which is a disadvantage. For example, if a device works in a virtualized Ubuntu, it does not automatically mean it will work in CentOS too (while both operating systems are based on the same Linux kernel). At the same time, the IOMMU unit (see 2.4) is needed for proper direct hardware assignment. The IOMMU takes care of memory protection and valid address space mapping. Perhaps the biggest stumbling block is that the direct assignment solution can't share one physical device associated with a virtual instance with other virtual instances. Mapping is usually 1 to 1 – one device per virtual instance. There is also a problem of device migration to other instances – the hypervisor does not control the communication between the device and the instance. Unfortunately, the hypervisor plays a key role to achieve correct device management and more security, if it's omitted, there is a definite loss of flexibility. Some of these problems are being addressed by the industry standard of PCI-Express hardware virtualization the SR-IOV.



(a) (b)

Figure 3.3: (a) Scheme of assigning I/O devices directly to a virtual instance. (b) The difference between direct assignment and hardware SR-IOV solution. Source [30].

Hardware virtualization technology SR-IOV

SR-IOV (*Single Root Input/Output Virtualization*) is a PCI hardware virtualization technology. It is released as an extension of the PCI standard from the PCI-SIG group, [25]. The specification defines that a PCI device (with SR-IOV support) is presented in the system as multiple PCI bus level devices (mentioned in 2.3). One primary device is always a physical function (PF) and the other available devices are independent virtual functions (VF). Each VF has its own PCI configuration space. Each VF also shares one or more physical resources on the network adapter, such as an external network port, with the PF and other VFs. Virtual functions have limited capabilities, but inherently they can serve as network devices. A VF is not a fully-fledged PCIe device. However, it provides a basic mechanism for directly transferring data between a VM child partition and the underlying SR-IOV network adapter. Software resources associated for data transfer are directly available to the VF and are isolated from use by the other VFs or the PF. The configuration of most of these resources is performed by the PF miniport driver that runs in the management operating system of the hypervisor. PF contains the SR-IOV control structure and it's used to manage a set of associated VFs. Each VF can be assigned to a virtual machine. In order for the device to work as a virtual network card, a switch/bridge must be implemented inside, this bridge forwards packets to different virtual instances (aka. different VFs). If

a PCI device has more than one VF, it must have sufficient hardware resources (buffers, interrupt lines, etc.) for each VF. There is a theoretical maximum limit on the number of VFs, it depends on physical device capability and resources. The difference between direct I/O device assignment and SR-IOV technology is shown in figure 3.3b.

Devices (VF) can be shared/migrated between multiple virtual instances and this is an indisputable advantage. For network devices, you can even connect multiple VFs as network lines, thereby aggregating (increasing) the transmission speed of the link. Unfortunately, the technology is also associated with several drawbacks such as: a specification is exclusively for PCIe devices, a limited number of VFs (fixed resource allocation), the complexity of a hardware solution (development of a new PCI device is costly and it depends heavily on hardware design) and a lack of composability – a hassle with device mappings to virtual instances. Mappings are closely linked to the hardware, and the hardware must be aware of the changes, otherwise the device will not adapt to it (mapping management is not purely software overhead). Complete SR-IOV scheme with all of the mentioned components (plus BDF is bus:device:function and Q means queue), is shown in figure 3.4.

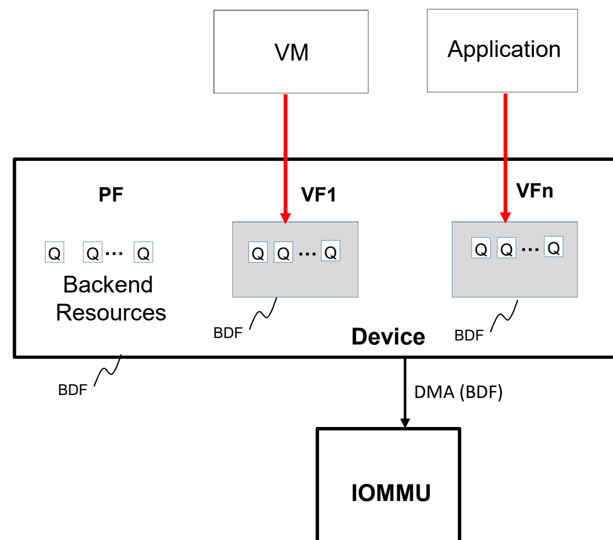


Figure 3.4: SR-IOV scheme. Source [32].

3.3 Scalable I/O virtualization – III. Generation

The latest trend in the I/O device virtualization is the combination of the two previous generations. Combining the benefits of hardware virtualization (high performance) with software flexibility (virtual device management). The goals are a flexible resource allocation – create a selectable number of virtual devices (as a user option) from a single physical device (as opposed to SR-IOV, where the hardware fixes the number of VFs, see 3.2), and more composability that comes with it naturally – created virtual devices can be dynamically assigned to virtual instances using software (hypervisor). The difference is that the management passes from hardware under the hypervisor. The goals of this generation are nicely outlined in figure 3.5 and an example scheme of how such a system might look is in figure 3.6a.

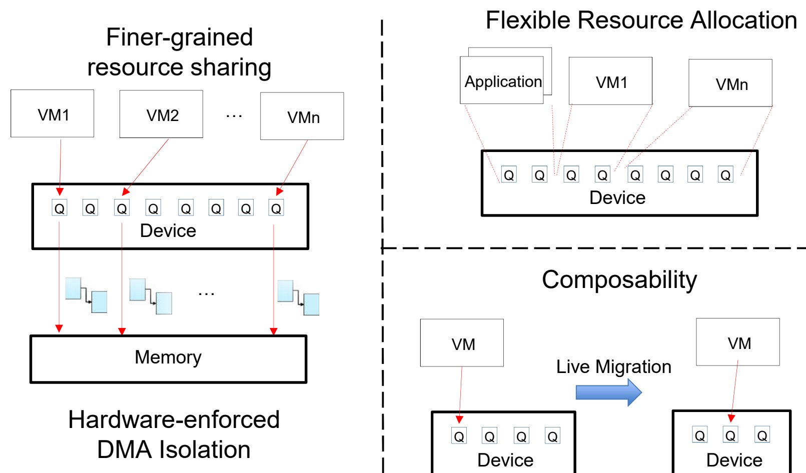
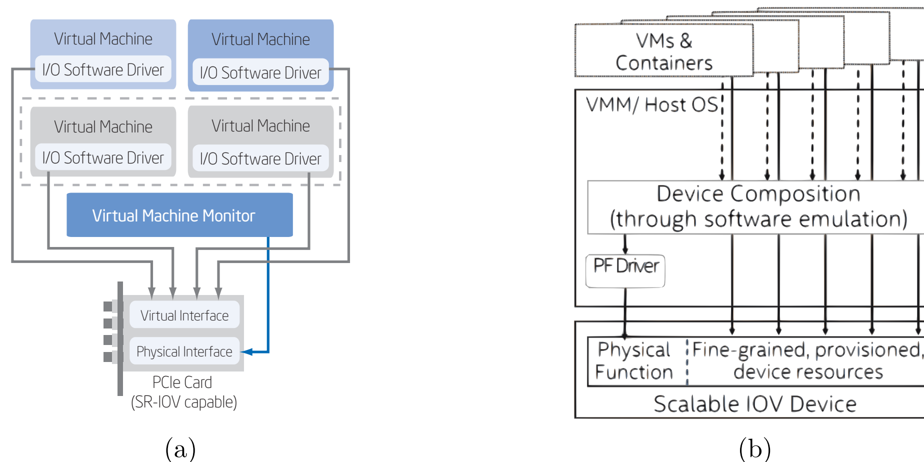


Figure 3.5: Goals of the scalable I/O virtualization. Source [32].



(a) (b)

Figure 3.6: (a) Direct assigning and sharing I/O devices using Intel® VT-d (IOMMU) with the SR-IOV device support. Source [9]. (b) Scalable I/O virtualization using replication and virtual device composition, diagram shows different data and control path.

Intel corporation is trying to push the presented concept as a standard into the IT industry, under the name Intel Scalable I/O Virtualization (using virtio interface). The concept in principle is using two different communication paths between physical hardware and a virtualized device. Data paths are directly connected from the physical device to the virtual instance and control paths are passed through the host hypervisor. Figure 3.6b illustrates this fact, the control paths are indicated by a dashed line – they lead to a hypervisor that has direct connection (via the driver) to the physical device and the data paths lead directly between the virtual instance and the device. This gives us good control of virtualized devices as well as fast data transmission – a combination of the advantages of I. and II. generation I/O virtualization. One of the essential hardware components bringing this together is the IOMMU unit (see 2.4), thanks to IOMMU it's possible to safely share virtual devices. Developments regarding III. Generation – scalable I/O device virtualization is an ongoing process (as of the time of writing this project), standards are not finished and

the implementation part (from a hardware and software point) is still within a design and prototyping stage. News and information can be found in sources [11, 36].

However, nothing prevents the creation of new projects aimed to implement new software that supports scalable I/O virtualization, even when specifications are still under development. One such project is the VFIO (*Virtual Function I/O*) software framework. The primary purpose of the VFIO is to offer a transport mechanism in a framework to driver developers working on direct device assignment in OS.

VFIO framework

The VFIO (*Virtual Function I/O*) software framework aims to solve the problem of direct device assignment, such that the user mode may use the DMA interface capabilities of the devices directly through the aid of the Virtual Device Composition Module (VDCM), figure 3.7. This allows the developer to manipulate easily with ADIs (Assignable Device Interfaces, ADI represents minimal sharable resources such as queues, queue pairs, contexts, and ADI is enumerated via DVSEC capability, 2.3). The ADIs are mapped into the system devices (in a case of SR-IOV as direct devices or in a form of mediated devices which are described later). In order to make the best use of the VFIO ability for enabling smooth DMA, there needs to be an IOMMU collaboration. The IOMMU must support PASID, and thus the translation is extended to bus:device:function:PASID instead of the classic BDF. The VDCM is not part of VFIO and has to be solved independently - there is no device management by VFIO.

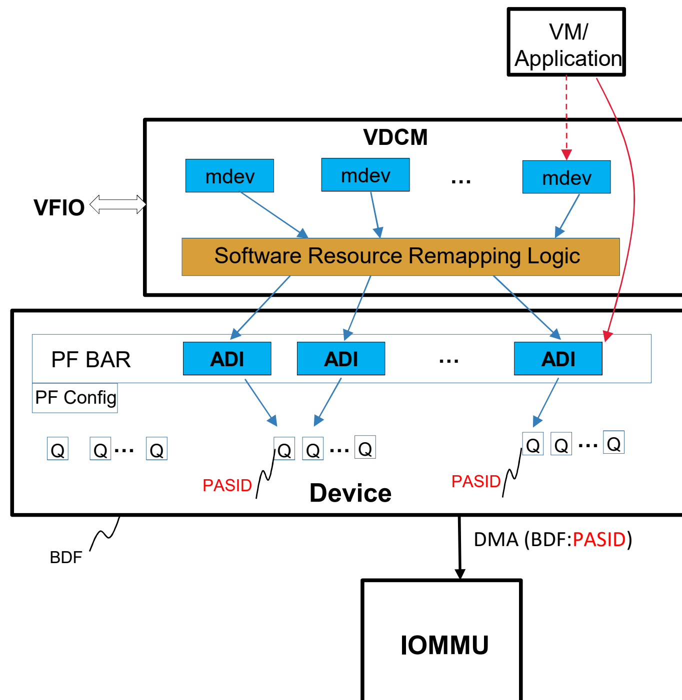


Figure 3.7: A basic device

In a KVM virtualized environment, there is an option to directly assign devices (using `pci-assign`[2]), figure 3.8 shows this. Because KVM is a hypervisor and not a general device

[2]https://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM

driver, this introduces some problems: access to device resources is not compatible with a secure boot (safe initialization), there isn't a guaranteed IOMMU granularity (IOMMU groups), there is an inadequate model for the ownership of the device (security reasons), it's exclusive only to x86 architecture, it supports only PCI bus devices and it works only in a KVM/QEMU virtualization environment.

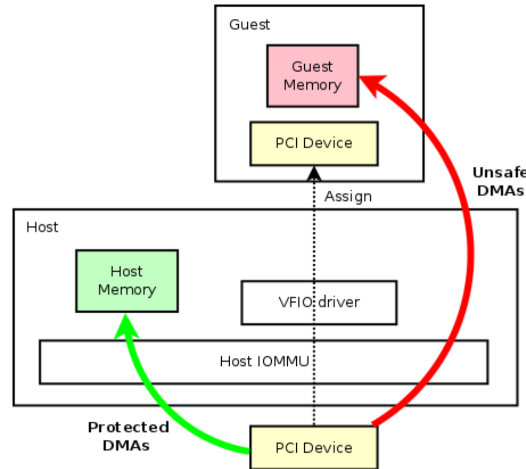


Figure 3.8: A basic device assignment use case in KVM/QEMU without VFIO.[3]

The VFIO is an inter-layered device driver inside kernel. VFIO directly uses an IOMMU driver to protect device access and restricts user mode access to the domain secured device. Sadly, the VFIO itself doesn't manage devices and can only pass the entire device (ADI) to a virtual instance, resulting in low device utilization (small flexibility, see problems in 3.2). To increase the efficiency of resource utilization SR-IOV virtualization technology (3.2) could be used. When using SR-IOV, there is one physical device and multiple virtual functions. VFs are isolated by PCIe ACS (Access Control Services), preventing them from accessing each other – ensuring stability, and through the use of VFIO it is possible to attach each VF (ADI) to a virtual instance.

Mediated devices, VFIO-mdev virtualized environment

Yet SR-IOV doesn't fix the challenge of creating dynamic VFs on the fly – VFs are hard-coded in the hardware firmware and hardware must support SR-IOV technology to work properly. But how can devices that do not explicitly support SR-IOV provide us with the same functionality as SR-IOV? Nvidia is proposing an extended *VFIO-mdev* [14] framework to solve this problem. The core of the mdev (*mediated device*) model is in the abstraction of the real hardware device state (by emulating it) and storing it into the data structure of the created mdev device. Thereafter, the mdev devices (with possibility of creating multiple different mdev devices from one physical device) are registered to the system scheduler as hardware devices and the scheduler will ensure time share multiplexing, so that the "physical" hardware devices can be shared between multiple virtual instances. Figure 3.9 represents a schematic diagram of the VFIO-mdev framework. The framework provides a set of device management interfaces.

[3]Source: https://wiki.qemu.org/Features/VT-d

Operations on mdev devices are as following:

1. Create or remove a mdev device.
2. Add or remove a mdev device to/from the mdev bus driver.
3. Add or remove a mdev device to/from the IOMMU group.

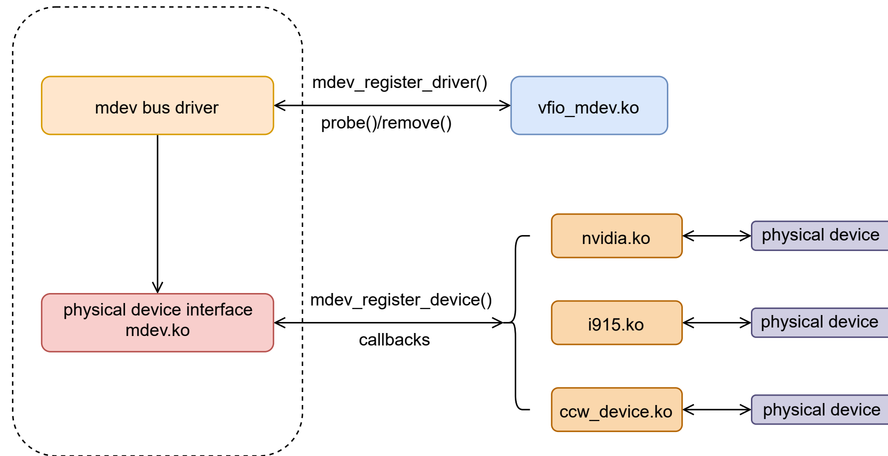


Figure 3.9: VFIO-mdev software framework.

With this concept, it is possible to build a mediated device from any device (only mdev drivers are required for that device) and then assign it to a virtual machine using the VFIO framework.

Chapter 4

Designing new device drivers

At the time of writing this thesis, there are no device drivers that can offer I/O virtualization for the NDK platform. This chapter includes a draft of two new device driver prototypes. The presented designs are based on theoretical foundations in chapters 2 and 3. The desired goals of the new drivers are to achieve the **best possible data throughput** (with a throughput in a non-virtualized environment) and **adequate flexibility**, manageability of newly created virtual devices. This chapter is separated into two parts. The first part (section 4.1) contains the design concept of a fully software-virtualized driver using `virtio-vhost` groundwork (falls into I. generation of I/O virtualization). This concept will fulfill the laid down goals, but there is an expected lack of performance due to emulation. For a theoretical improvement, another design is proposed in the second part (section 4.2). Instead of using pure software emulation, this concept is based on the mix of latest virtualization technologies, namely VFIO-mdev framework (falls into III. generation of I/O virtualization, for theory see 3.3) which only emulates device configuration space but accelerates data transfers by direct device passthrough.

4.1 Vhost software driver

The driver requirements are the possibility of dynamically creating new virtual I/O devices, which means creating new interfaces with different configurations, e.g. multiple Tx/Rx channels (transmit/receive) per device, etc., and proper maintenance of the newly created devices. There is also the need for each virtual interface to achieve adequate transfer rates - throughput. Initially, as the first prototype in iterative development, the software virtualization (emulation/paravirtualization) seems to be the best and probably the simplest solution.

There are several solutions, but, as seen in the 3.1 section, there already exists an unified virtual virtio interface for data transfers between the virtual machine and the device. Let's take advantage of this and build a driver on the top of it. Virtio has a number of basic implementations (see figure 3.2a), but neither one of these implementations fits for our use case. The reason is that up to three copies of the data exist in the simplest virtio implementation (in the context of network traffic; application data [virtual instance] → virtio front-end driver [virtual instance] → virtio back-end driver [hypervisor] → device driver, physical I/O space [hardware]). All these copies are slowing down/reducing the resulting throughput. The virtio is just a transmission standard and therefore it does not prevent anybody to implement new mechanisms over the plain virtio-virtqueues. There is one such

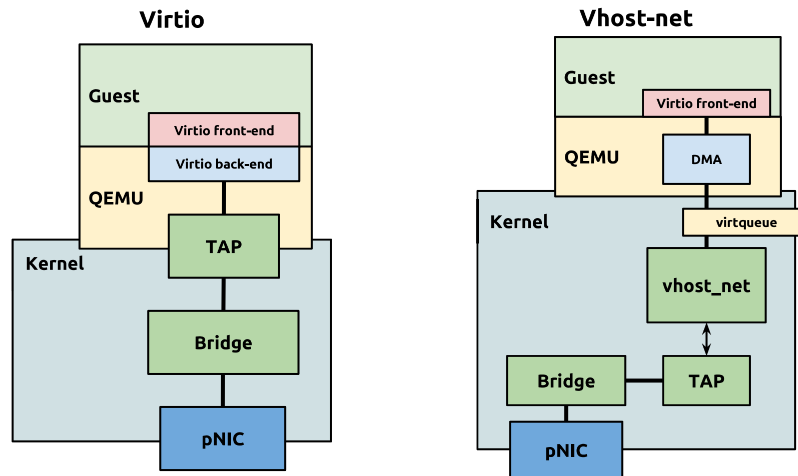


Figure 4.1: The difference between a pure virtio and the vhost-net architecture. Source [30].

network interface driver implementation, the *vhost-net* set of drivers. The vhost-net driver particularly supports DMA transfers (using interrupt signalling) and as a result of that it allows faster transmissions (up to an 8-fold increase[1]) versus basic virtio implementations. The difference between a pure virtio and a vhost architecture is represented in figure 4.1.

As described in a section 2.5, we know that the `nfb` driver is responsible for card initialization and data transfers via a PCIe bus between the physical FPGA card and the Linux operating system. So if we combine NDK platform with vhost architecture and place it above the `nfb` driver, we create a new driver `npp_vhost` – see 4.2 scheme. Red parts are indicating changes in the NDK drivers. Change in the `nfb` driver consists only of a method for creating multiple new internal nfb devices that will be assigned to virtual instances. Another change is made in the original vhost-net driver[2] itself, by making a fork of vhost-net and rewriting parts of code for `nfb` integration (the core and internal operations of vhost are retained) into the new `npp_vhost` driver. The driver `npp _vhost`[3] is internally connected to the `nfb` driver (to be able to convert created nfb devices to vhost devices) and externally connected to `virtio-net` VM driver. The connection is implemented via a virtio-virtqueue data path. Inside the virtual instance, `virtio-net` creates a standard Linux network device (e.g., eth0) over which applications work and can send their data. In scheme (4.2), all solid black lines are data flow and dashed lines are information channels. In order to achieve better throughput, the vhost can use DMA (at least one data copy is reduced), the interrupt system is used to signal DMA transfers. This set of drivers should offer us excellent management of virtual devices and sufficient transfer rates. If we want to create a new I/O virtual device, we simply send a request to the `npp_vhost` (e.g., through the hypervisor's libvirt control) to create the device and assign it to a virtual instance (flexibility – migration of devices is achieved similarly). The entire software emulation takes place exclusively in the `npp_vhost` driver. The driver must transform all operations of a vhost-net device into the nfb device (internal NDK platform device) operations.

[1] https://www.linux-kvm.org/page/UsingVhost

[2] vhost-net source code: https://elixir.bootlin.com/linux/latest/source/drivers/vhost/net.c

[3] npp is an abbreviation in slovak language for netcope paketové prenosy, english translation - netcope packet transmissions

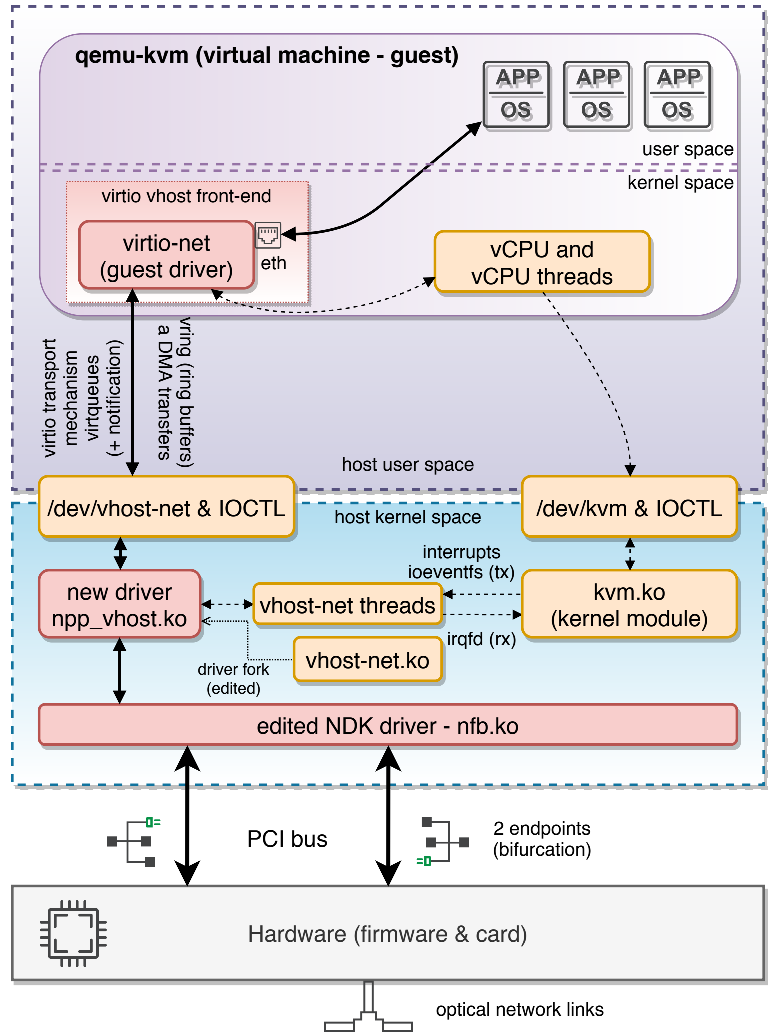


Figure 4.2: Detailed schematic design of the new `npp_vhost` driver over the NDK platform for I/O virtualization.

4.2 VFIO-mdev software driver

This section presents a more complex driver design. The driver provides virtually the same flexibility as the previous design, but will also provide a substantial improvement in data throughput. The 4.3 scheme shows the new design. The proposed framework is based on the VFIO and mdev technologies mentioned in section 3.3, hence the name – VFIO-mdev driver. In principle, this framework should achieve speeds comparable to SR-IOV virtualization (i.e., native hardware speed without virtualization). This brings the question: Why not use the standardized SR-IOV? Presented framework has the advantage that it doesn't need to enforce SR-IOV and it in theory get the "same" results, in theory. To fully implement SR-IOV in firmware is much more difficult than do something similar in software (it's harder to create, verify and test hardware firmware, rather than a software implementation). All the red blocks shown in figure 4.3 need to be modified to deploy the VFIO-mdev system into the NDK platform. The required changes in the `nfb` driver (the driver in the host) are related to creation of new internal NFB devices, similar to the vhost

driver (4.1). The nfb driver running in a virtual instance doesn't need to be modified as the mdev device emulation enables the driver to see the NFB PCIe card attached to the system (in the guest) as if it was a true physical device. The most complicated modification is that the vfio-mdev system driver has to be re-implemented in order to accomplish the nfb device emulation, IOMMU device control and DMA proper initialization. The VFIO-mdev inspiration comes from an example of Nvidia's original virtual PCI serial mediated tty (TeleTYpewriter) driver[4]. The vfio-mdev driver is responsible for creating a new mdev device (see 3.3), assigning the mdev device to the correct IOMMU group (see 2.4), registering the mdev device into the mdev bus and emulation related to device management - providing PCI configuration memory space (see 2.5), interrupt handling, etc.

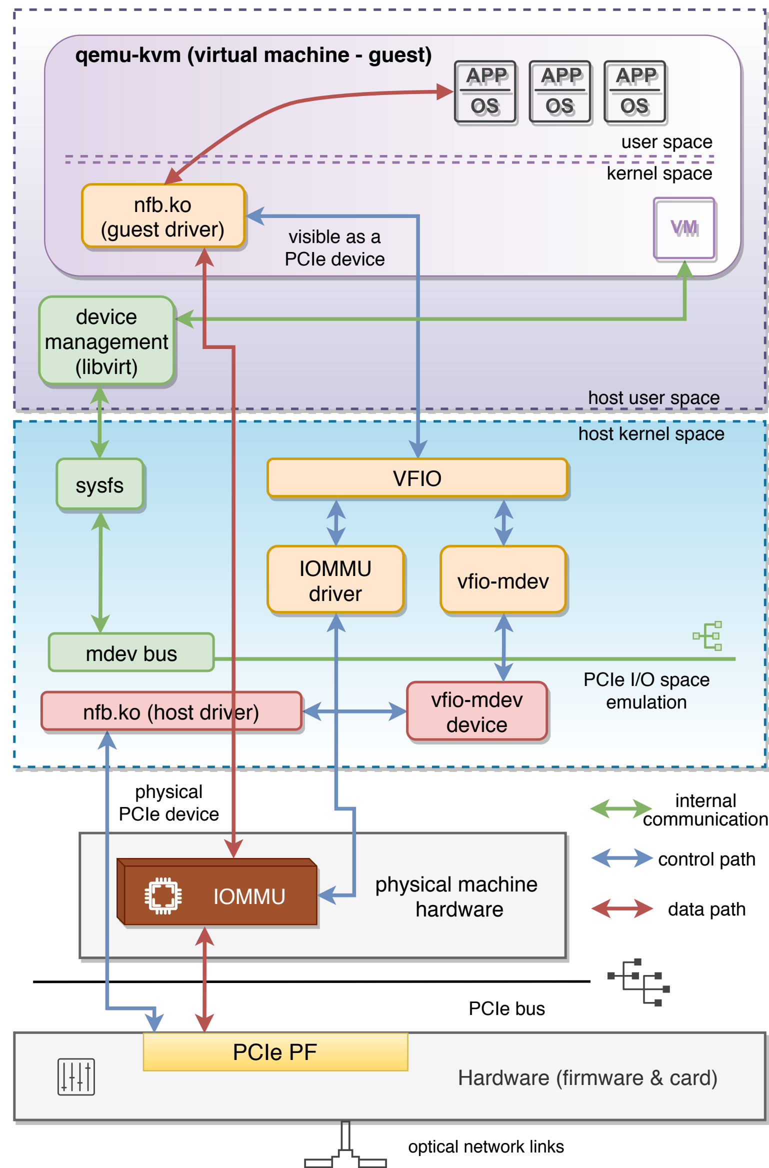


Figure 4.3: Detailed schematic design of the new vfio-mdev driver over the NDK platform for I/O virtualization.

[4]mtty source code: https://elixir.bootlin.com/linux/latest/source/samples/vfio-mdev/mtty.c

Each mdev device need to be registered within the IOMMU (the IOMMU group/domain need to be assigned). When all the mdev devices are in the same IOMMU group/domain with the original physical device, they will be able to communicate and share memory space with each other. Of course as a matter of principle, only physical devices share memory space (virtualized devices have the same resources as the physical device). By using the emulation of I/O memory space, the `nfb` driver running in a virtual instance is able to see the NFB device at its allocated addresses for that particular VM (guest), another nfb driver in another instance has differently allocated NFB device addresses which may even physically overlap. The IOMMU converts each virtual instance's virtual device addresses to known physical I/O addresses. Due to this principle it is possible to use DMA transfers between a virtual instance (mdev device) and a physical device. The use of VFIO-mdev establishes distinct pathways in the system architecture – divided into data and control paths. Solid red lines show the data paths in scheme 4.3 (direct pass-through - DMA, no hypervisor interception). The software handles the control paths (solid blue lines) mainly by emulation of the true physical device.

Chapter 5

Implementation

In this chapter, we plan to discuss the current state of the implementation, outline the challenges that have arisen in attempting to fulfill the criteria identified in the previous chapters, define the methods that have been used to solve them and illustrate some interesting elements in the development process. At the beginning of this chapter section 5.1 addresses the state of the `npp_vhost` driver. This solution has been unsuccessful and makes the driver unusable due to performance issues in the current configuration. Luckily, the second design based on the VFIO-mdev framework, the `mdev_nfb` driver seems to offer better possibilities to achieve desired goals, section 5.2.

5.1 npp_vhost driver

Our objectives are simple, to provide a kernel driver capable of delivering almost native data throughput (between VM and host) while at the same time offering the possibility of creating scalable (flexible) virtual network interfaces from a software point of view. In fact, one such approach could be easily implemented, as described in chapter 4.1. The vhost method as a complete software emulation is well known and several device implementations from hard drives to network cards are already in use inside Linux kernel. One particular, the vhost-net was chosen as the basis for the driver development.

vhost-net changed into npp_vhost driver

The main sources on which this section is built are [19, 34]. The virtio specification is based on two elements: devices and drivers. In a typical implementation, the hypervisor exposes the virtio devices to the guest through a number of transport methods. By design they look like physical devices connected to the guest within the virtual machine. The most common transport method is PCI or PCIe bus. When the guest boots and uses the PCI/PCIe auto discovering mechanism, the virtio devices identify themselves with the PCI vendor ID and their PCI Device ID. The guest's kernel uses these identifiers to know which driver must handle the device. In particular, the linux kernel already includes virtio drivers. The virtio drivers must be able to allocate memory regions that both the hypervisor and the devices can access for reading and writing, i.e., via memory sharing. The data plane is that part which uses these memory regions, and a control plane is the process of setting them up. The virtio kernel drivers share a generic transport-specific interface (e.g: virtio-pci – virtqueue mechanism as guest kernel module driver), used by the actual transport and device implementation (such as virtio-net, or virtio-scsi).

The vhost protocol (vhost API) is a message based protocol that allows the hypervisor to offload the data plane to another component (handler) that performs data forwarding more efficiently. Using this protocol, the master sends the following configuration information to the handler:

- The hypervisor memory layout. This way, the handler can locate the virtqueues and buffer within the hypervisor memory space.
- A pair of file descriptors that are used for the handler to send and receive the notifications defined in the virtio specification.

After this process, the hypervisor will no longer process packets (read or write to/from the virtqueues). Instead, the dataplane will be completely offloaded to the handler, which can now access the virtqueue's memory region directly as well as send and receive notifications directly to and from the guest. The vhost messages can be exchanged in any host-local transport protocol, such as Linux sockets or character devices.

The vhost-net is a kernel driver that implements the handler side of the vhost protocol to implement an efficient data plane, i.e., packet forwarding. In this implementation, qemu and the vhost-net kernel driver (handler) use ioctls (input/output control, a system calls for device-specific input/output operations) to exchange vhost messages and a couple of eventfd-like file descriptors called irqfd and ioeventfd are used to exchange notifications with the guest. When vhost-net kernel driver is loaded, it creates a character device in sysfs – `/dev/vhost-net` and then when qemu is launched with vhost-net support it opens it and initializes the vhost-net instance with several `ioctl` calls. These are necessary to associate the hypervisor process with the vhost-net instance, prepare for virtio feature negotiation and pass the guest physical memory mapping to the vhost-net driver. During the initialization the vhost-net kernel driver creates a kernel thread called `vhost-$pid`, with hypervisor process PID. This thread is called the "vhost worker thread".

The tap device is still used to connect the VM to the host, but now the worker thread handles the I/O events i.e., it polls for driver notifications or tap events and forwards data. Qemu allocates one eventfd and registers it to both vhost and KVM in order to achieve the notification bypass. The `vhost-$pid` kernel thread polls it and KVM writes to it when the guest writes data to a specific address. This mechanism is named *ioeventfd*. This way a simple read/write operation to a specific guest memory address does not need to go through the expensive QEMU process wakeup and can be routed to the vhost worker thread directly. This also has the advantage of being asynchronous, no need for the vCPU to stop (no need of an immediate context switch).

The prototype driver has been implemented (source code is included as part of this thesis) – almost all parts of original `vhost-net` were preserved and parts that have been added to change it to `npp_vhost` were mainly an overlay of TUN/TAP socket device (nothing exceptional for an in-depth analysis, it's also worth bearing in mind that the source code is just a messy prototype). Unfortunately, the initial testing revealed poor performance results and so the development was moved to a more promising `mdev_nfb` driver. The cause of poor performance is attributed to the polling system. This particular method is not suitable for the NDK platform, since the `nfb` device does not have (not supported) an interrupt mechanism in its current version, and so the vhost-net implementation must always use a "slow" polling method, no matter what. Polling method is still using *ioeventfd* of the emulated TUN/TAP device, but instead of interrupts it is using fallback timers to produce fake interrupts if nothing new occurs, all this is happening with the additional cost

of context switching. Perhaps a bit of tinkering with it might improve performance, but instead of spending time on it, focus has been shifted towards a second proposed driver design.

5.2 mdev_nfb driver

The goals are still the same, but as described in the previous section, the implementation has been moved to the VFIO-mdev architecture due to driver performance problems. The concept of VFIO-mdev itself was explored in chapter 4.2 – the key components of the concept are mdev devices (mdev framework), VFIO framework and IOMMU. With that being said, the core of the practical implementation begins with the Nvidia mtty sample code [15, 14], which has been modified into the final `mdev_nfb` driver.

mtty sample code - mediated device

The base-level of the entire Linux kernel system consists of many directories with different roles and functions (drivers, cryptography features, memory management, etc.). One of these directories is devoted to prototypes for new technologies for demonstration and experimentation – the `samples` directory. Inside this directory[1] is a dummy driver program to demonstrate how to use the mediated device framework inside the kernel, the `vfio-mdev/mtty.c`. Specifically, this driver's purpose is to create an mdev device which simulates a serial port over a "PCI device". The details on how the driver works are as follows:

1. Build and load the mtty.ko sample kernel module.
   This step creates a dummy device `/sys/devices/virtual/mtty/mtty/` and files in this device directory in sysfs should look like this:

```
# tree /sys/devices/virtual/mtty/mtty/
    /sys/devices/virtual/mtty/mtty/
    |-- mdev_supported_types
    |    |-- mtty-1
    |    |    |-- available_instances
    |    |    |-- create
    |    |    |-- device_api
    |    |    |-- devices
    |    |    \-- name
    |    \-- mtty-2
    |         |-- available_instances
    |         |-- create
    |         |-- device_api
    |         |-- devices
    |         \-- name
    |-- mtty_dev
    |    \-- sample_mtty_dev
    |-- power
    |    |-- autosuspend_delay_ms
    |    |-- control
    |    |-- runtime_active_time
    |    |-- runtime_status
    |    \-- runtime_suspended_time
    |-- subsystem -> ../../../../class/mtty
    \-- uevent
```

[1] https://elixir.bootlin.com/linux/latest/source/samples/vfio-mdev

2. Create a mediated device by using the dummy device that is created in the previous step:

```
# echo "84a8f4f2-dead-beef-3c1e-e6b00b5a1001" > \
    /sys/devices/virtual/mtty/mtty/mdev_supported_types/mtty-1/create
```

3. Add parameters of the new device to VM guest, for example with libvirt (virsh) update qemu-kvm environment VM XML configuration:

```
<hostdev mode='subsystem' type='mdev' managed='no' model='vfio-pci'>
    <source>
        <address uuid='84a8f4f2-dead-beef-3c1e-e6b00b5a1001'/>
    </source>
</hostdev>
```

4. Boot the VM and the device inside the guest will appear on the PCI bus:

```
# lspci -s 00:##.# -vv
  00:##.# Serial controller: Device 4348:3253 (rev 10)
          Subsystem: Device 4348:3253
          Physical Slot: 5
          Control: I/O+ Mem- BusMaster- SpecCycle- MemWINV-
          Region 0: I/O ports at c150 [size=8]
          ...
```

5. To destroy created mediated device, simply do:

```
# echo 1 >\
    /sys/bus/mdev/devices/84a8f4f2-dead-beef-3c1e-e6b00b5a1001/remove
```

   or unload kernel module (as root):

```
# rmmod mtty
```

The driver emulates the entire device. It is accomplished by imitating the entire PCI configuration space. A piece of source code shown in listing 5.1 sets "predefined" values of virtual configuration space for emulated device.

For example notice the line 4 in the source code, the driver sets PCI device ID value at address `0x0` to `0x32534348` (as stated in chapter 2.3) and the consequence of this is that VM sees inside a guest operating system a PCI device identified as `Subsystem:Device`

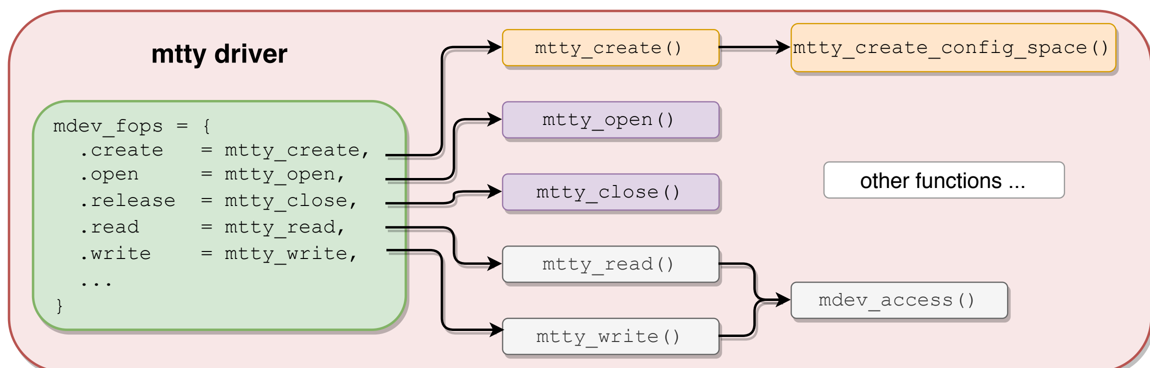


Figure 5.1: Schematic of the `mtty` driver functions and their interaction.

```
1  static void mtty_create_config_space(struct mdev_state *mdev_state)
2  {
3      /* PCI dev ID */
4      STORE_LE32((u32 *) &mdev_state->vconfig[0x0], 0x32534348);
5      /* Control: I/O+, Mem-, BusMaster- */
6      STORE_LE16((u16 *) &mdev_state->vconfig[0x4], 0x0001);
7      /* Status: capabilities list absent */
8      STORE_LE16((u16 *) &mdev_state->vconfig[0x6], 0x0200);
9      /* Rev ID */
10     mdev_state->vconfig[0x8] =  0x10;
11     // ...
12     /* base address registers */
13     /* BAR0: IO space */
14     STORE_LE32((u32 *) &mdev_state->vconfig[0x10], 0x000001);
15     mdev_state->bar_mask[0] = ~(MTTY_IO_BAR_SIZE) + 1;
16     /* Subsystem ID */
17     STORE_LE32((u32 *) &mdev_state->vconfig[0x2c], 0x32534348);
18     mdev_state->vconfig[0x34] =  0x00;   /* Cap Ptr */
19     mdev_state->vconfig[0x3d] =  0x01;   /* interrupt pin (INTA#) */
20     /* Vendor specific data */
21     mdev_state->vconfig[0x40] =  0x23;
22     mdev_state->vconfig[0x43] =  0x80;
23     // ...
24 }
```

Listing 5.1: Code snippet of `mtty_create_config_space()` function which fills `vconfig` structure with predefined values.

`4348:3253`[2]. Also observe the lines 14, 15 where a similar process is happening, only this time it sets the whole peripheral I/O memory to I/O type with specified size (BAR0, ref 2.3). But this is only a partial step for correct device emulation.

The complete `mtty` driver overview of the functions and their relations is shown in figure 5.1. Another important part is the proper management of the write/read requests directed onto a "virtual" device to/from guest VM. To even think about handling requests, first the registration of the driver operations must be carried out. Inside the mdev framework exists a mediated device operations structure, which is used to register all available device functions into the framework (basics of mdev framework is described in 3.3). In our example the `mdev_fops` structure is shown at the top of the listing C.1. The driver needs to implement these operations and register them into the framework by invoking the `mdev_register_driver()` function with the full `mdev_fops` structure as a parameter.

Mediated device framework support operations such as:

- creating and removing the dummy sysfs device (i.e., step 1 in the mtty sample module; `/sys/devices/virtual/mtty/mtty/`, the dummy device is created only once, per module lifetime)
- opening and closing (releasing) a mediated device (i.e., step 2 in the mtty sample module; it processes parameters, in this case UUID[3] and the driver will create new internal structure to work with; in general, multiple mediated devices with different

[2] output of `lspci` command inside guest

[3] Universally Unique IDentifier, https://en.wikipedia.org/wiki/Universally_unique_identifier

dynamics can be created – each device could have unique memory sizes, etc., it all depends on what is actually implemented by the driver itself)

- read and write requests going to the "virtual" mdev device (e.g., calling `lspci` inside VM will activate internal kernel pci module functions for PCI devices, the VFIO framework will capture these requests and pass them to the mdev framework, the mdev framework via `mdev_fops` will determine actual implementation and call corresponding function with given parameters to process it.)

  One such example of a specific function for handling read requests is shown in the listing C.1. Whenever VFIO handles a request (between guest and host), it transfers it to → mdev framework. Inside the mdev framework, our read `mtty_read()` function is registered through `mdev_fops`. Based on the specified parameters, it will call the `mdev_access()` function, which will process the parameters and, in this case, copy the emulated data from the vconfig configuration structure back to the VFIO-mdev buffer, which will be returned to the VM guest. This shows the main principle, similar actions also applies for write requests, etc.

As have been described read/write accesses are managed in the VFIO context. One aspect is to emulate these accesses, but if we need to have access to the actual I/O device - mainly direct memory access, and then transfer the request results back to the host VM via VFIO, the VFIO process must operate closely with IOMMU.

The VFIO and IOMMU interoperability

Key references in the formation of this section are [17, 13, 37, 29]. Devices are the main target of any I/O driver. Devices usually create a programming interface consisting of I/O control, interrupts and DMA. Without getting through the specifics of all of these, DMA is by far the most critical component of ensuring a secure environment, as it allows read-write access to system memory, these device accesses to the system represent the greatest risk to the overall integrity of the system. In order to help mitigate this risk, many modern IOMMUs (see 2.4) now incorporate isolation properties into what was, in many cases, a translation-only interface (i.e., addressing problems for devices with limited address spaces). With this, devices can be isolated from each other and from arbitrary memory access, thus allowing things like secure direct assignment of devices into virtual machines. This isolation is not always at the granularity of a single device. Even if IOMMU is capable of doing so, the properties of devices, interconnects and IOMMU topology may reduce this isolation. For instance, an individual device may be part of a larger multi-function enclosure. Although the IOMMU may be capable of distinguishing between devices inside the enclosure, the enclosure may not require transactions between devices to reach the IOMMU. Example of this could be anything from a multi-function PCI device with backdoors between functions to a non-PCI-ACS (Access Control Services) capable bridge allowing redirection without reaching the IOMMU. Topology can also play a factor in terms of hiding devices (see 2.3). A PCIe-to-PCI bridge masks the devices behind it, making transaction appear as if from the bridge itself. Obviously IOMMU design plays a major factor as well.

For the most part an IOMMU may have a device level granularity, but any system could be susceptible to reduced granularity on device level. The IOMMU API therefore supports the concept of IOMMU groups. A group is a set of devices that can be isolated from all other devices in the system. As a consequence, groups are the unit of ownership used by

VFIO/IOMMU. Although the group is the minimum granularity that must be used to ensure secure user access, it's not necessarily the preferred granularity. In IOMMUs which make use of page tables, it may be possible to share a set of page tables between different groups, reducing the overhead both to the platform (reduced TLB thrashing, reduced duplicates in page tables) and to the user (programming only a single set of translations). For this exact reason the VFIO makes use of a container class, which may hold one or more groups. A container is created by simply opening the `/dev/vfio/vfio` character device. On its own, the container provides little functionality. The user needs to add a group into the container for the next level of functionality. To do this, the user first needs to identify the group associated with the desired device. This can be done using the sysfs links (for pci devices, `/sys/bus/pci/devices/XXXX`). By unbinding the device from the host driver and binding it to a VFIO driver, a new VFIO group will appear as `/dev/vfio/$GROUP`, where $GROUP is the IOMMU group number of which the device is a member. If there are multiple devices in one IOMMU group, each device must be bound to the VFIO driver before operations in the VFIO group are allowed.

- Find the default IOMMU group assigned at the start of the system (after the boot if an IOMMU is enabled, it automatically adds devices to separate groups) for a specific device e.g.:

```
# cat /sys/bus/pci/devices/0000:04:0b.0/iommu_group
  42
```

- The device is in IOMMU group 42. It's a PCI device (meaning it's on the PCI bus), therefore the user has to make use of the vfio-pci driver to manage the group:

```
# modprobe vfio-pci
```

- Unbind the device from the current driver, then bind it to the vfio-pci driver. This creates a new VFIO group character device for this group:

```
# lspci -n -s 0000:04:0b.0
  04:0b.0 0404: dead:beef (rev 12)
# echo 0000:04:0b.0 > /sys/bus/pci/devices/0000:04:0b.0/driver/unbind
# echo dead beef > /sys/bus/pci/drivers/vfio-pci/new_id
# ls /dev/vfio/42
```

Once the group is ready, it may be added to the container by opening the VFIO group character device (`/dev/vfio/$GROUP`) and using the `VFIO_GROUP_SET_CONTAINER` ioctl, passing the file descriptor of the previously opened container file. If required and supported by the IOMMU driver to share the IOMMU context between groups, several groups may be set to the same container. When a group cannot be assigned to the current group container, a new empty container would need to be used instead. With a group (or groups) added to a container, the other ioctls are available to allow access to VFIO IOMMU interfaces. Additionally, it becomes possible to get file descriptors for each device within a group using an ioctl on the VFIO group file descriptor. The VFIO device API includes device ioctls for I/O regions and their read/write/mmap offsets on the device descriptor, as well as mechanisms for identifying and registering interrupt notifications. Following listing 5.2 shows how to use VFIO API to work with containers.

```
int container, group, device, i;
struct vfio_group_status group_status   = { .argsz = sizeof(group_status) };
struct vfio_iommu_type1_info iommu_info = { .argsz = sizeof(iommu_info) };
struct vfio_iommu_type1_dma_map dma_map = { .argsz = sizeof(dma_map) };
struct vfio_device_info device_info     = { .argsz = sizeof(device_info) };
container = open("/dev/vfio/vfio", O_RDWR);          // Create a new container
if (ioctl(container, VFIO_GET_API_VERSION) != VFIO_API_VERSION) // Unknown API version
if (!ioctl(container, VFIO_CHECK_EXTENSION, VFIO_TYPE1_IOMMU))
    /* Doesn't support the IOMMU driver we want. */
group = open("/dev/vfio/42", O_RDWR);                // Open the group
ioctl(group, VFIO_GROUP_GET_STATUS, &group_status);  // Test the group is viable
if (!(group_status.flags & VFIO_GROUP_FLAGS_VIABLE))
    /* Group is not viable (i.e., not all devices bound for vfio) */
ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);  // Add the group to the container
ioctl(container, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU);  // Enable the IOMMU model that we want
ioctl(container, VFIO_IOMMU_GET_INFO, &iommu_info);  // Get additional IOMMU info
/* Allocate some space and then setup a DMA mapping */
dma_map.vaddr = mmap(0, 1024*1024, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
dma_map.size = 1024*1024;
dma_map.iova = 0; // 1MB starting at 0x0 from device view
dma_map.flags = VFIO_DMA_MAP_FLAG_READ|VFIO_DMA_MAP_FLAG_WRITE;
ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);
device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:04:0b.0"); // Get a device file descr
ioctl(device, VFIO_DEVICE_GET_INFO, &device_info);  // Test and setup the device
for (i = 0; i < device_info.num_regions; i++) {
    struct vfio_region_info reg = { .argsz = sizeof(reg) };
    reg.index = i;
    ioctl(device, VFIO_DEVICE_GET_REGION_INFO, &reg);
    /* Setup mappings... read/write offsets, mmaps
       For PCI devices, config space is a region */
}
for (i = 0; i < device_info.num_irqs; i++) {
    struct vfio_irq_info irq = { .argsz = sizeof(irq) };
    irq.index = i;
    ioctl(device, VFIO_DEVICE_GET_IRQ_INFO, &irq);
    /* Setup IRQs... eventfds, VFIO_DEVICE_SET_IRQS */
}
```

Listing 5.2: Code snippet of the VFIO API that performs ioctl calls over VFIO IOMMU containers.

The question arises, why do we need VFIO containers and what are they good for? The answer is clear, VFIO containers manage IOMMU and we need it for the DMA control, otherwise our mediated device would not be able to perform DMA transfers (it is the most critical task). Notice lines 3, 15 and 16 in listing 5.2, these lines display indirect work with the IOMMU, such as setting the container explicitly to the `VFIO_TYPE1_IOMMU` type. Type1 IOMMU, often referred to as x86 IOMMU is a backend driver for IOMMU and it's designed for the physical hardware AMD-Vi & Intel VT-d IOMMU implementation inside kernel [29]. Theoretically, it is possible to re-use and implement similar IOMMUs such as SPARC or other architectures over this backend, but let us stick to x86 instead. Type1 IOMMU supports the IOMMU API and has little, if any, limitations across the IOVA range that can be mapped (due to the involvement of IOMMU, the physical address the hardware uses may not be the real physical address, but instead a completely arbitrary input-output virtual address – IOVA assigned to the hardware by the IOMMU). The Type1 IOMMU

is currently optimized for relatively static mappings of a userspace process with userspace pages pinned into memory. Type1 IOMMU also implies the devices and IOMMU domains are PCI oriented as the IOMMU API is focused on the device/bus interface rather than the group interface.

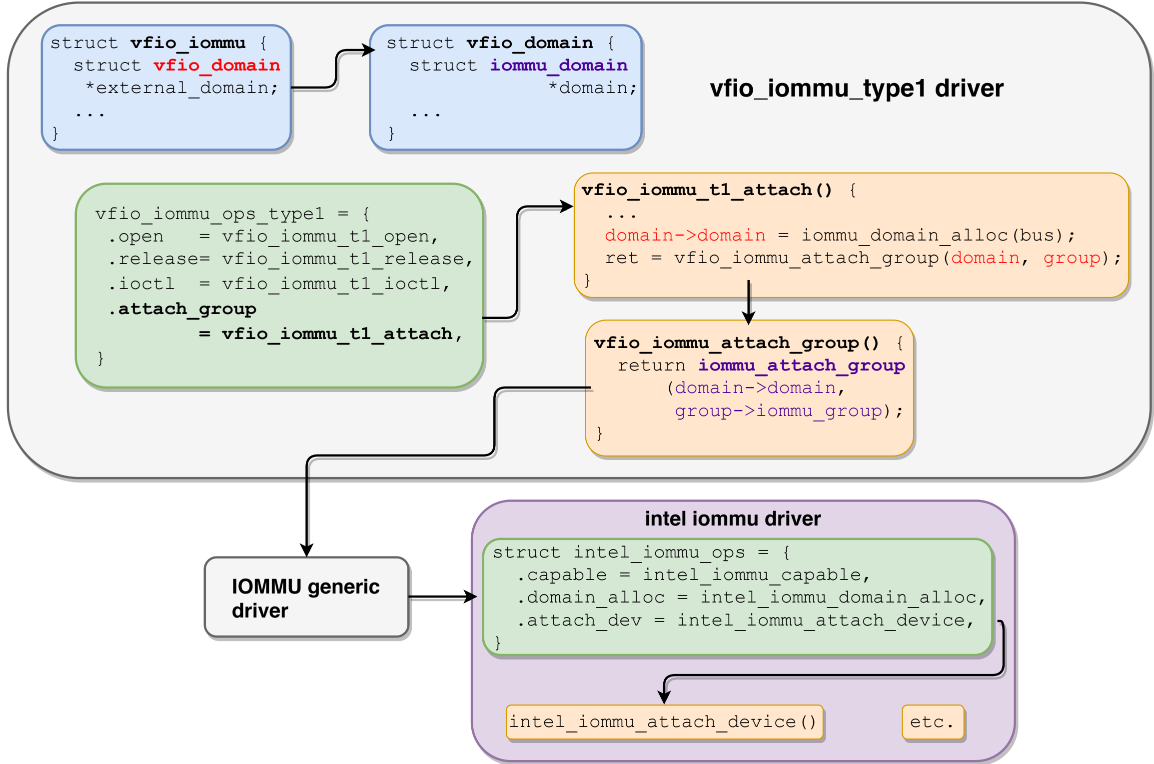


Figure 5.2: Schematic of the `vfio_iommu_type1`, the relationship between the structures, functions and their interactions.

Listing C.2 displays the source code of /drivers/vfio/vfio_iommu_type1.c driver, figure 5.2 shows the relationship between the structures and functions inside and outside the driver, the interlayer - generic IOMMU driver and the vendor's specific implementation, i.e. Intel IOMMU. The red and purple colors inside the figure distinguish between vfio and iommu devices (physical iommu). Snippets (C.2) contain `vfio_iommu` and `vfio_domain` structures which relate to the currently used IOMMU x86 driver (Intel/AMD). The VFIO framework itself follows the standard kernel driver operations concept — in this case `vfio_iommu_driver_ops`, the `vfio_iommu_type1_attach_group()` function is registered in the ops structure. Whenever an external event (user) attempts to attach a device to a group (in the sense of an IOMMU group attachment), it invokes the mentioned function, inside this function is the hard-coded instance of the `iommu_attach_group()` function that is actually called (it depends on the specific hardware whether Intel IOMMU or AMD IOMMU is recognized by system, inside the system is again used similar concept of the `iommu_ops`, e.g., `intel_iommu_ops = .attach_dev = intel_iommu_attach_device`, more implementation details in [29]).

In theory, once the device is finally in the IOMMU group (inside the VFIO container) and the group settings are set correctly, the DMA transactions, our core objective, should be fully functional. Unfortunately, it only works for "real" hardware devices connected

to a VFIO framework (for example, if the SR-IOV is used, 3.2). It's not that simple for mediated devices, because mdev is just an emulated physical device that isn't directly connected to the IOMMU, the parent device is (physical hw). The `VFIO_IOMMU_TYPE1` has been implemented to link the mediated device to the same IOMMU group as the parent device. When mdev and the parent device are in different groups, DMA transactions will fail with denied access, IOMMU isolation would be violated. Alas, putting devices into the same group still doesn't guarantee the correct functionality, it depends on the parent device, it might work, but the isolation will still be compromised, one mdev would be able to access the address space of another mdev and that's not our desired goal. As written in the kernel mailing list [2, 3], **vfio/mdev: IOMMU aware mediated device from 22 Feb 2019**:

> " *The Mediate Device is a framework for fine-grained physical device sharing across the isolated domains. Currently the mdev framework is designed to be independent of the platform IOMMU support. As the result, the DMA isolation relies on the mdev parent device in a vendor specific way.*
>
> *There are several cases where a mediated device could be protected and isolated by the platform IOMMU. For example, Intel VT-d rev3.0 [10] introduces a new translation mode called 'scalable mode', which enables PASID–granular translations. The VT-d scalable mode is the key ingredient for Scalable I/O Virtualization [11, 33] which allows sharing a device in minimal possible granularity (ADI - Assignable Device Interface).*
>
> *A mediated device backed by an ADI could be protected and isolated by the IOMMU since 1) the parent device supports tagging an unique PASID to all DMA traffic out of the mediated device; and 2) the DMA translation (IOMMU) unit supports the PASID granular translation. We can apply IOMMU protection and isolation to this kind of devices just as what we are doing with an assignable PCI device.*
>
> *In order to distinguish the IOMMU-capable mediated devices from those which still need to rely on parent devices, this patch set adds one new member in struct mdev_device.*
>
> *The mdev parent device driver could opt-in that the mdev could be fully isolated and protected by the IOMMU when the mdev is being created by invoking mdev_set_iommu_device() in its @create().*
>
> *On IOMMU side, a basic requirement is allowing to attach multiple domains to a PCI device if the device advertises the capability and the IOMMU hardware supports finer granularity translations than the normal PCI Source ID based translation.*
>
> *As the result, a PCI device could work in two modes: normal mode and auxiliary mode. In the normal mode, a pci device could be isolated in the Source ID granularity; the pci device itself could be assigned to a user application by attaching a single domain to it. In the auxiliary mode, a pci device could be isolated in finer granularity, hence subsets of the device could be assigned to different user level application by attaching a different domain to each subset.*
>
> *Below APIs are introduced in iommu generic layer for aux-domain purpose:*
> * `iommu_dev_has_feature(dev, IOMMU_DEV_FEAT_AUX)`

- *Detect both IOMMU and PCI endpoint devices supporting the feature (aux-domain here) without the host driver dependency.*

∗ `iommu_aux_attach_device(domain, dev)`

- *Attaches @domain to @dev in the auxiliary mode. Multiple domains could be attached to a single device in the auxiliary mode with each domain representing an isolated address space for an assignable subset of the device.*

∗ `iommu_aux_get_pasid(domain, dev)`

- *Return ID used for finer-granularity DMA translation. For the Intel Scalable IOV usage model, this will be a PASID. The device which supports Scalable IOV needs to write this ID to the device register so that DMA requests could be tagged with the right PASID prefix. "* [2, 3]

That is precisely what we need. Some mechanism (auxiliary domains) which ensures automatic isolation (not relying on the parent device, but on the IOMMU), by using PASID (see 2.3 and 2.4).

How it all fits together

5.2 describes how to create a mediated device and how to allocate it to a VM, e.g. through libvirt (virsh tool). The qemu-kvm system secures a device connection to the VFIO framework and the VFIO framework sets all the requirements for correct IOMMU usage (`VFIO_IOMMU_TYPE1`; using the `ioctl` calls, described in 5.2), this whole process is automatic. If the system supports IOMMU auxiliary domains, the IOMMU will use the `PASID` feature to ensure device isolation (in DMA translation for each device), otherwise the system must rely on the parent device to handle the translation and isolation. Here's a stumbling block of this project, the NDK platform doesn't contain any driver that controls IOMMU, meaning that everything is under automatic system management. It's very hard to try to implement any IOMMU control, certain aspects of IOMMU are concealed from the common user and unique to the vendor, there's no open specification for IOMMU (close cooperation/partnership with the manufacturer is required - Intel/AMD). Also, the IOMMU API (in kernel) is still under development and new features are appearing and adjusting on a relatively regular basis. If we don't use the auxiliary domains, the DMA won't work (most likely memory mapping will fail) unless we use some nasty code hacks to completely ignore the isolation and proper IOMMU operation (applying the hacks could cause system instability, etc., more on hacks later in the evaluation section 6.2).

```
1  #define ecap_pasid(e)     ((e >> 40) & 0x1)
2  #define ecap_smts(e)      (((e) >> 43) & 0x1)
3
4  #define sm_supported(iommu)     (intel_iommu_sm && ecap_smts((iommu)->ecap))
5  #define pasid_supported(iommu)  (sm_supported(iommu) && ecap_pasid((iommu)->ecap))
6
7  static bool intel_iommu_dev_has_feat(struct device *dev, enum iommu_dev_features feat) {
8      if (feat == IOMMU_DEV_FEAT_AUX) {
9          int ret;
10         if (!dev_is_pci(dev) || dmar_disabled ||
11             !scalable_mode_support() || !iommu_pasid_support()) {
```

```
12              return false;
13          }
14          ret = pci_pasid_features(to_pci_dev(dev));
15          if (ret < 0) {
16              return false;
17          }
18          return !!siov_find_pci_dvsec(to_pci_dev(dev));
19      }
20      return false;
21  }
```

Listing 5.3: Intel IOMMU driver source code in the kernel, the requirements that need to be fulfilled for scalabe virtualization support.

Another option is to use the `PASID` (aux domains). Sadly, the present situation is such that AMD does not officially support VFIO mediated devices using PASID (AMD IOMMU in general supports PASID, but not in a mechanism viable for more advanced virtualization techniques). As has been discussed, Intel has a dedicated Scalable Virtualization specification where a PASID usage is listed [10, 11]. Listing 5.3 displays code from the Intel IOMMU driver inside the kernel where support for scalable virtualization appears to be available. There are several conditions that must be met in order to use `PASID` in the system. First of all, the device must be a PCI device and the PCI device itself must support the generation of PASID tags within PCI transactions (just emulate a PCI PASID capability isn't enough, to recall look at 2.3 and check appendix B).

The IOMMU in the system has to be able to process PASID and needs to support scalable mode (each IOMMU hardware has a list of capabilities, equivalent to PCI, values in these registers determine the functionality provided by the vendor). To check IOMMU PASID capability we need to inspect extended capabilities (ecap) register in IOMMU's DMAR (DMA remapping) unit. For example:

```
# cat /sys/class/iommu/dmar0/intel-iommu/ecap
  f020df -> 1111 0000 0010 0000 1101 1111 (24 bits)
```

In this case, the processor doesn't support the PASID nor scalable mode (upper bits are all zeroes, in specification the PASID is 40th and scalabe mode 43rd bit inside ecap register).

Overview of changes made to the `nfb_mdev` driver

Inside the previous sections, it has been outlined how the mtty (a.k.a. mediated device) is implemented, the fundamentals of the VFIO framework have been defined with the integration of an IOMMU. All that's left is to put it all together. There is only one minor change in the original nfb module (`/swbase/drivers/kernel/drivers/nfb/`, driver version 6.6.0), and that is an increase in the maximum number of secondary drivers that may be attached to the nfb kernel core, listing 5.4. Warning: the shown source codes (only snippets) are incomplete, the purpose of these snippets is to explain the basic concepts.

```
1  #define NFB_DRIVERS_MAX     8
2  nfb_registered_drivers[index] = ops;
3  int nfb_driver_register(struct nfb_driver_ops ops) {
4      for (i = 0; i < NFB_CARD_COUNT_MAX; i++) {
5          if (nfb_devices[i])
6              nfb_attach_driver(nfb_devices[i], index);
```

```
7       }
8   }
9   void nfb_attach_driver(struct nfb_device* nfb, int i) {
10      if (nfb_registered_drivers[i].attach && nfb->list_drivers[i].status ==
        ↪   NFB_DRIVER_STATUS_NONE) {
11          ret = nfb_registered_drivers[i].attach(nfb, &nfb->list_drivers[i].priv);
12          nfb->list_drivers[i].status = ret == 0 ? NFB_DRIVER_STATUS_OK :
            ↪   NFB_DRIVER_STATUS_ERROR;
13      }
14  }
```

Listing 5.4: Changes made in the original `nfb` module driver, driver version 6.6.0.

Changes had to be made in the modular compilation of the drivers as well. The Makefile has been updated with a new module object and a new folder has been added to the drivers subdirectory - `mdev/` directory, where all the files of the `nfb_mdev` driver are located. Listing 5.5 captures these changes and new folder structure.

```
1   CONFIG_NFB_MDEV          = m
2   obj-$(CONFIG_NFB_MDEV)  += mdev/
3   $> tree mdev/
4      mdev/
5      +--- fdt
6      +--- Makefile
7      +--- mdev_nfb.c
8      +--- mdev_nfb.o
9      +--- modules.order
10     +--- Module.symvers
11     \--- nfb_mdev.ko
```

Listing 5.5: Updates in driver compilation structure and new `mdev` folder.

Figure 5.3 shows the composition of the `nfb_mdev` driver, and there are certain similarities with `mtty` (see figure 5.1). The main and only source code is `mdev_nfb.c`. It is a copy of the mentioned `mtty.c`, 5.2. The structure of the code is preserved as it was in the original. There are no special changes in a mediated device API (`mdev_parent_ops`), functions that are implemented:

- `.create = mdev_nfb_create(),`
- `.remove = mdev_nfb_remove(),`
- `.read = mdev_nfb_read(),`
- `.write = mdev_nfb_write(),`

For the kernel module itself, the functions

- `module_init(mdev_nfb_init())`
- `module_exit(mdev_nfb_exit())`

are implemented. For instance, the `mdev_nfb_init()` function registers the module to the nfb driver and then registers (creates) the mediated device from its parent device (from nfb pci device), as is shown in the listing 5.6.

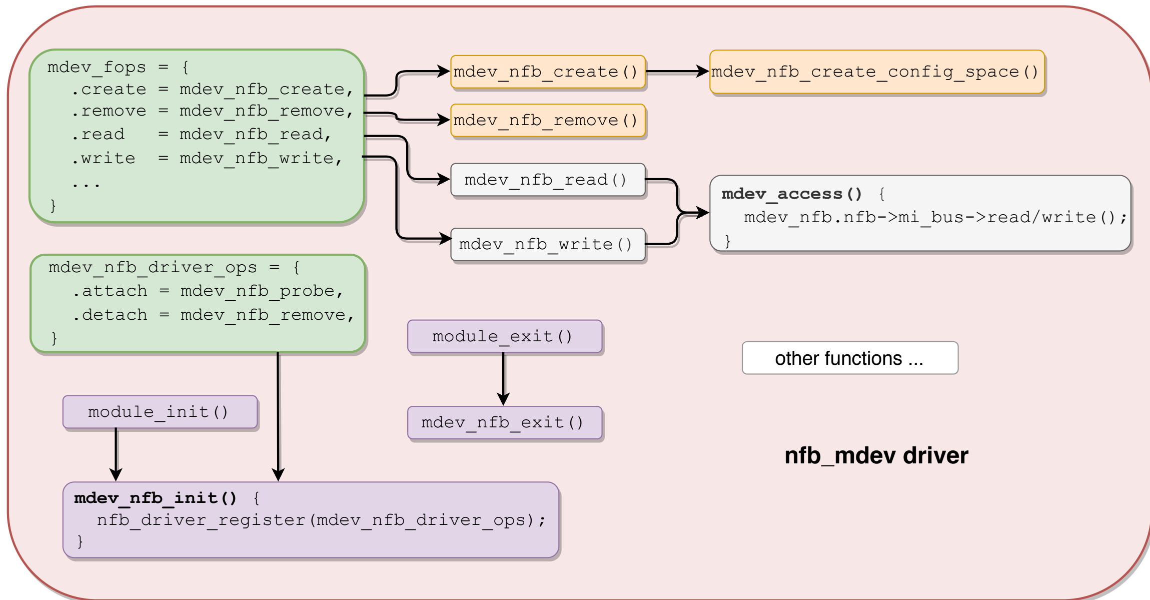


Figure 5.3: The composition of the `nfb_mdev` driver, showing the interactions, operations and functions.

```
1  static int __init mdev_nfb_init(void) {
2      nfb_driver_register(mdev_nfb_driver_ops);
3      // finding only first possible nfb device
4      for (i = 0; i < MAX_MDEV_NFB_DEVICES; ++i)
5          if (nfb_ptrs[i] != NULL)
6              break;
7      if (i == MAX_MDEV_NFB_DEVICES) {
8          pr_info("%s: didn't found nfb device !\n", __func__);
9          return -1;
10     } else
11         pr_info("%s: found at least one nfb device [%d]\n", __func__, i);
12     memset(&mdev_nfb, 0, sizeof(mdev_nfb));
13     mdev_nfb.nfb = nfb_ptrs[i];
14     mdev_nfb.dev = (nfb_ptrs[i]->pci->dev);
15     mdev_register_device(&(mdev_nfb.nfb->pci->dev), &mdev_fops);
16 }
```

Listing 5.6: `mdev_nfb_init()` implementation inside `nfb_mdev` driver.

After successful module initialization and registration into the mdev API, the framework itself automatically initiates the creation of a mediated device, via `mdev_nfb_create()` function, listing 5.7. This function contains FDT (*Flattened Device Tree*) manipulation of original nfb (see DT usage in [26]), then generation of virtual space that emulates mediated PCI device configuration space and all other necessities connected to mdev initialization.

```
1  int mdev_nfb_create(struct kobject *kobj, struct mdev_device *mdev) {
2      struct mdev_state *mdev_state;  // structure to work with -> repr. mdev internal state
3      /* create new fdt */
4      fdt = kzalloc(NFB_FDT_MAXSIZE, GFP_KERNEL);
5      ret = fdt_create_virtual_tree(fdt);
```

```
6       // find RX/TX channels in parent nfb device and copy properties to 'emulated' device
7       fdt_for_each_compatible_node(mdev_nfb.nfb->fdt, node, "netcope,dma_ctrl_sze_tx") {
8           count_tx++;
9       }
10      ret = fdt_copy_channel_node_props(mdev_nfb.nfb->fdt, fdt, mdev_nfb.open, TX_DIR);
11      // initiate mdev_state
12      mdev_state = kzalloc(sizeof(struct mdev_state), GFP_KERNEL);
13      mdev_state->fdt = fdt;  // set fdt
14      mdev_state->irq_index = -1;
15      mdev_state->vconfig = kzalloc(MDEV_CONFIG_SPACE_SIZE, GFP_KERNEL); // alloc conf. space
16      mutex_init(&mdev_state->ops_lock);
17      mdev_state->mdev = mdev; // set mdev
18      mdev_set_drvdata(mdev, mdev_state); // set driver data, internal mdev state
19      mdev_nfb_create_config_space(mdev_state); // create an emulated PCI conf. space for
        ↪ mdev
20  } // ... mdev_nfb_create_config_space
21  int mdev_nfb_create_config_space() {
22      /* Set Subsystem ID based of nfb PCI original */
23      STORE_LE16((u16 *) &mdev_state->vconfig[0x2c], mdev_nfb.nfb->pci->subsystem_vendor);
24      STORE_LE16((u16 *) &mdev_state->vconfig[0x2e], mdev_nfb.nfb->pci->subsystem_device);
25      mdev_state->vconfig[0x34] =  0xc0;   /* Cap Ptr */
26      mdev_state->vconfig[0x3d] =  mdev_nfb.nfb->pci->pin;   /* interrupt pin (INTA#) */
27      /* Vendor specific data */
28      // PCI-Express capability emulation (starting at address 100h)
29      mdev_state->vconfig[0xc0] = 0x10;
30      mdev_state->vconfig[0xc1] = 0x00;
31      mdev_state->vconfig[0xc2] = 0x00;
32      mdev_state->vconfig[0x100] =  0x0b; // ...
33  }
```

Listing 5.7: `mdev_nfb_create()` function, along with the `mdev_nfb_create_config_space()` implemented function.

The function `mdev_nfb_create_config_space()` creates almost one to one PCI configuration space of an nfb device, keeps Subsystem Vendor and Device IDs the same, sets PCI to PCIe extended capability (PCIe for larger memory space) and it also preserves other values of original registers. At this point, within the source code, the mdev device should be successfully created in the sysfs (i.e., `/sys/class/mdev_bus/0000:3b:00.0/nfb/nfb0/device/mdev_supported_types/nfb-mdev_default/`). If someone wants to start working with mdev (recall 5.2, the UUID `echo` command), that means they're trying to open a device descriptor.

The last but not least functionality of the driver is the handling of read/write requests. The method is pretty much the same as in `mtty` (read/write splitted into access function), the difference is in accessing the actual physical hardware. To do this, it's using AXI MI bus as si shown in the listing 5.8, details in [7, 8].

```
1  if (offset > 10000) {
2      res = mdev_nfb.nfb->mi_bus->write(mdev_nfb.nfb->mi_bus, buf, count, offset);
3      break;
4  } else {
5      handle_bar_write(index, mdev_state, offset, buf, count);
6  }
7  // ...
8  if (offset > 10000) {
9      res = mdev_nfb.nfb->mi_bus->read(mdev_nfb.nfb->mi_bus, buf, count, offset);
```

```
10        break;
11    } else {
12        handle_bar_read(index, mdev_state, offset, buf, count);
13    }
```

Listing 5.8: nfb_mdev driver read/write requests implementation, processed via nfb MI bus.

This section summarizes the details of the mdev_nfb driver implementation. In addition to allowing the virtualization of mediated devices, the driver needs to work entirely with system drivers such as vfio, vfio_iommu_type1 and mdev module drivers.

Chapter 6

Evaluation

This chapter discusses the approach that was used for testing the implemented system. Another goal is to outline the author's experience with used technologies, and provide some guidelines for future projects of this type.

6.1 Evaluation

The Liberouter organization's internal servers have been used for the research analysis, development, testing of new technologies, and evaluating implemented drivers. The primary server used in the above operations has been `cider.liberouter.org`, running on Scientific Linux kernel version 5.4.7-1.el7.elrepo with an x86_64 Intel architecture - Intel(R) Xeon(R) Silver 4114 CPU. The NDK FPGA card available in the machine was NFB-200G2QL. In order to fully check the functionality of the implemented driver, we need to reboot the system with our own PASID-implemented design (B). Listing 6.1 captures the task of flashing the FPGA card with a new firmware design.

```
cider(SL7) ~$ uname -r
5.4.7-1.el7.elrepo.x86_64
cider(SL7) ~$ nfb-boot -f 0 ~/pasid_capability/fwbase/applications/nic/nfb-200g2ql/nfb-200
    g2ql_nic.nfw
Bitstream size: 30732412 B (118 blocks)
Erasing Flash: 100% [=================================================================]
Writing Flash: 100% [=================================================================]
cider(SL7) ~$ nfb-info
--------------------------------------- Board info ----
Card name                  : NFB-200G2QL
Serial number              : 53
Network interfaces         : 2
------------------------------------ Firmware info ----
Project name               : NIC_200G2QL_100GE
Built at                   : 2020-02-20 11:41:41
Build tool                 : Vivado v2019.1.1 (64-bit)
Build author               : xperes00@stud.fit.vutbr.cz
RX queues                  : 4 (only 2 available)
TX queues                  : 4 (only 2 available)
ETH channels               : 2
-------------------------------------- System info ----
PCI slot                   : 0000:3b:00.0
NUMA node                  : 0
cider(SL7) ~$ sudo lspci -v -s 3b:
```

```
3b:00.0 Ethernet controller: Netcope Technologies, a.s. NFB-200G2-master
        Subsystem: Netcope Technologies, a.s. Device 0800
        Flags: bus master, fast devsel, latency 0, IRQ -1, NUMA node 0
        Memory at ac000000 (64-bit, non-prefetchable) [size=64M]
        Capabilities: [40] Power Management version 3
        Capabilities: [70] Express Endpoint, MSI 00
        Capabilities: [100] Advanced Error Reporting
        Capabilities: [140] Single Root I/O Virtualization (SR-IOV)
        Capabilities: [180] Alternative Routing-ID Interpretation (ARI)
        Capabilities: [1a0] Device Serial Number 41-89-13-65-00-a3-d1-00
        Capabilities: [1c0] #19
        Capabilities: [480] Vendor Specific Information: ID=0d7b Rev=1 Len=020 <?>
        Capabilities: [4a0] Process Address Space ID (PASID)
        Kernel driver in use: nfb
        Kernel modules: nfb
```

Listing 6.1: Flashing the NFB-200G2QL card with our own desing.

As we can see, the card has been correctly flashed and booted with a new configuration, we can also see that the lspci tool demonstrates the enabled PASID capability of firmware design - Capabilities: [4a0] System Address Space ID (PASID). Next step is to remove the default system initialized nfb driver and insert our nfb driver instead. Subsequently, initializate all the other drivers required for proper mediated device virtualization, drivers such as vfio, vfio_iommu_type1, mdev and vfio_mdev. We can check the modules initialization output with dmesg command. The listing 6.2 shows how it has to be achieved.

```
cider(SL7) ~$ rmmod nfb
cider(SL7) ~$ insmod ~xperes00/Work_mdev/swbase/drivers/kernel/drivers/nfb/nfb.ko
cider(SL7) ~$ insmod /usr/src/kernels/5.4.7-1.el7.elrepo/drivers/vfio/vfio.ko
cider(SL7) ~$ insmod /usr/src/kernels/5.4.7-1.el7.elrepo/drivers/vfio/vfio_iommu_type1.ko
cider(SL7) ~$ insmod /usr/src/kernels/5.4.7-1.el7.elrepo/drivers/vfio/mdev/mdev.ko
cider(SL7) ~$ insmod /usr/src/kernels/5.4.7-1.el7.elrepo/drivers/vfio/mdev/vfio_mdev.ko
cider(SL7) ~$ dmesg
[ 232.223980] nfb 0000:3b:00.0: nfb_boot: Attached successfully
[ 232.224612] nfb 0000:3b:00.0: nfb_ndp: Attached successfully (4 RX and 4 TX DMA channels)
[ 232.296124] nfb 0000:3b:00.0: nfb_qdr: Attached successfully (0 QDR controllers)
[ 232.296127] nfb 0000:3b:00.0: successfully initialized
[ 233.527131] VFIO - User Level meta-driver version: 0.3
```

Listing 6.2: Initialization of the required system device drivers.

After the system drivers initialization part, we initialize our own nfb_mdev driver and then create a new mediated device using some random UUID, procedure how to do so is shown in listing 6.3.

```
cider(SL7) ~$ insmod ~xperes00/Work_mdev/swbase/drivers/kernel/drivers/mdev/nfb_mdev.ko
cider(SL7) ~$ sudo sh -c " echo "83b8f4f2-509f-382f-3c1e-e6bfe0fa1001" > /sys/class/
    mdev_bus/0000\:3b\:00.0/nfb/nfb0/device/mdev_supported_types/nfb-mdev_default/create"
cider(SL7) ~$ dmesg
[ 279.624788] mdev_nfb_init: mdev initializating nfb driver
[ 279.624793] mdev_nfb_probe()
[ 279.624795] mdev_nfb_init: found at least one nfb device [0]
[ 279.624823] nfb 0000:3b:00.0: MDEV: Registered
[ 321.148019] mdev_nfb_create, creation succeeded for mdev: 83b8f4f2-509f-382f-3c1e-
    e6bfe0fa1001
[ 321.148310] nfb 0000:3b:00.0: Adding to iommu group 34
[ 321.148407] vfio_mdev 83b8f4f2-509f-382f-3c1e-e6bfe0fa1001: Adding to iommu group 34
```

```
[ 321.148410] vfio_mdev 83b8f4f2-509f-382f-3c1e-e6bfe0fa1001: MDEV: group_id = 34
```

Listing 6.3: `mdev_nfb` driver initialization and mediated device creation.

At this point, the mdev has been essentially created and we could work with it. It is obvious from the diagnostic messages (dmesg), that the device is registered to the IOMMU group 34, which is a strong indicator that the device should be properly managed by IOMMU (but not necessary). The entire initialization is done automatically by the system. Next step is to setup a virtual machine to work with mdev. This may be achieved with the virt-manager software which has a user interface (figure D.1, D.2), or using the virsh bash tool in the terminal (listing 6.4). Attention: For correct PCIe device virtualization, we definitely need to use q35 virtualized chipset instead of i440fx, otherwise the VM does not recognize the device as PCIe, but as PCI. The Q35 chipset offers PCI topology changes, PCI-passthrough functionality, and other improvements that may be useful.

```
cider(SL7) ~$ virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     centos6.10                     shut off
 -     centos7.0                      shut off
 -     test-vf1                       shut off
 -     test-vf2                       shut off
 -     ubuntu18.04                    shut off
 -     ubuntu18.04-2                  shut off
 -     vm1                            shut off

cider(SL7) ~$ virsh edit vm1
# internal editor opened # (for example vim)
. . .
<hostdev mode='subsystem' type='mdev' managed='no' model='vfio-pci' display='off'>
    <source>
       <address uuid='83b8f4f2-509f-382f-3c1e-e6bfe0fa1001'/>
    </source>
    <address type='pci' domain='0x0000' bus='0x04' slot='0x00' function='0x0'/>
</hostdev>
. . . :wq

Domain vm1 XML configuration changed.
```

Listing 6.4: Virtual machine setup with virsh tool.

After setting up the mdev in a virtual machine, what's left to do is to turn the VM on. The VM system will boot and ask us to log in, listing 6.5. Quick test if everything is alright, try lspci to verify whether the mediated device is visible in PCI bus. As is shown in listing 6.5, the mdev is successfully active in the system and has also been correctly identified as Ethernet controller, Netcope Technologies, a.s. NFB-200G2-master.

```
cider(SL7) ~$ virsh start vm1 --console
# start of the virtual machine and automatic switch into guest console
[  OK  ] Started Crash recovery kernel arming.
[  OK  ] Started Realm and Domain Configuration.

CentOS Linux 8 (Core)
Kernel 4.18.0-147.el8.x86_64 on an x86_64

Activate the web console with: systemctl enable --now cockpit.socket
```

```
localhost login: centos
Password:
Last login: Sun Apr 24 05:26:35 on ttyS0
[centos@localhost ~]$ su
Password:
[root@localhost centos]# lspci -v -x -s 04:
04:00.0 Ethernet controller: Netcope Technologies, a.s. NFB-200G2-master
        Subsystem: Netcope Technologies, a.s. Device 0800
        Physical Slot: 0-2
        Flags: bus master, fast devsel, latency 0, IRQ 22
        Memory at ec000000 (64-bit, non-prefetchable) [size=64M]
        Memory at f0000000 (32-bit, non-prefetchable) [size=1M]
        Capabilities: [c0] Express Endpoint, MSI 00
        Capabilities: [100] Vendor Specific Information: ID=0d7b Rev=2 Len=020 <?>
        Kernel driver in use: nfb
        Kernel modules: nfb
00: 26 1b 50 c2 07 00 10 00 00 00 00 02 00 00 00 00
10: 04 00 00 ec 00 00 00 00 00 00 00 f0 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 26 1b 00 08
30: 00 00 00 00 c0 00 00 00 00 00 00 00 0b 01 00 00
```

Listing 6.5: Starting the VM and verify if mdev is available in PCI bus.

To test a read/write from/to the card (testing functionality of the card accesses not the mdev requests, listing 5.8), the `nfb` driver needs to be used inside the guest VM. To do so, the NDK software kit must be installed inside the guest VM, as is shown in listing 6.6.

```
[root@localhost centos]# git clone git@gitlab.liberouter.org:ndk/swbase.git
Cloning into 'swbase'...
Enter passphrase for key '.ssh/id_rsa':
remote: Enumerating objects: 976, done.
remote: Counting objects: 100% (976/976), done.
remote: Compressing objects: 100% (299/299), done.
remote: Total 6695 (delta 637), reused 736 (delta 553), pack-reused 5842
Receiving objects: 100% (6695/6695), 1.37 MiB | 0 bytes/s, done.
Resolving deltas: 100% (4117/4117), done.
[root@localhost centos]# cd swbase
[root@localhost swbase]# ./configure
[root@localhost swbase]# make install
```

Listing 6.6: Installing the NDK software kit - drivers, libraries and tools to work with `nfb` card.

The Ethernet layer of the card must be enabled before we start checking the DMA. If we don't enable it, the tests won't be relevant, because whether or not the transaction passed to the card, it will be interrupted by the PCS/PMA layer inside the card, so we couldn't be sure whether there was a VM $\rightarrow$ server problem or a host $\rightarrow$ card problem. Listing 6.7 displays how to perform this action.

```
cider(SL7) ~$ nfb-eth -e1
cider(SL7) ~$ nfb-eth
----------------------------- Ethernet interface 0 ----
Speed                        : 100 Gb/s
Transceiver status           : OK
Transceiver cage             : QSFP-0
------------------------------------- RXMAC Status ----
RXMAC status                 : ENABLED
Link status                  : UP
HFIFO overflow occurred      : False
```

```
Received octets            :                    0
Processed                  :                    0
Received                   :                    0
Erroneous                  :                    0
Overflowed                 :                    0
------------------------------------- TXMAC Status ----
TXMAC status               : ENABLED
Transmitted octets         :                    0
Processed                  :                    0
Transmitted                :                    0
Erroneous                  :                    0
Repeater status            : Normal (transmit data from application)

----------------------------- Ethernet interface 1 ----
Speed                      : 100 Gb/s
Transceiver status         : OK
Transceiver cage           : QSFP-1
------------------------------------- RXMAC Status ----
RXMAC status               : ENABLED
Link status                : UP
HFIFO overflow occurred    : False
Received octets            :                    0
Processed                  :                    0
Received                   :                    0
Erroneous                  :                    0
Overflowed                 :                    0
------------------------------------- TXMAC Status ----
TXMAC status               : ENABLED
Transmitted octets         :                    0
Processed                  :                    0
Transmitted                :                    0
Erroneous                  :                    0
Repeater status            : Normal (transmit data from application)
```

Listing 6.7: Enabling ethernet PCS/PMA layer of the card.

After successfully configuring mdev and installing guest VM drivers it is time to verify whether the `nfb_mdev` "emulation" host card really works. To verify the mdev's Device Tree, simply use the `dtc` tool, see listing D.1. The Device Tree values should be accessible through sysfs `/dev/nfb0`. If the RX/TX queues values match the host card's original values then the interface emulation should operate properly. This also means that our guest `nfb` driver will be properly attached, otherwise the driver will enable its fallback mode which is undesirable. For example, the values of the physical card: `dma_module@0x01000000{ dma_ctrl_sze_rx0{ reg = <0x1000000 0x40>;} }` are identical to the mdev's values inside the guest VM. To verify if the guest `nfb` driver is really attached in the guest VM check the `dmesg` output, listing 6.8. Notice that one RX and one TX queue have been successfully recognized, the current version of `nfb_driver` enables exactly 1 RX/TX.

```
[root@localhost centos]# dmesg
[   7.577332] nfb 0000:04:00.0: NetCOPE FDT loaded.
[   7.580426] nfb 0000:04:00.0: nfb_mi: Attached successfully
[   7.580450] nfb 0000:04:00.0: nfb_boot: No boot_controller found in FDT.
[   7.580741] nfb 0000:04:00.0: nfb_ndp: Attached successfully (1 RX and 1 TX DMA channels)
[   7.634987] nfb 0000:04:00.0: nfb_qdr: Attached successfully (0 QDR controllers)
[   7.634990] nfb 0000:04:00.0: successfully initialized
[root@localhost centos]# nfb-info
--------------------------------------- Board info ----
Network interfaces         : 0
```

```
------------------------------------ Firmware info ----
RX queues                   : 1
TX queues                   : 1
ETH channels                : 0
-------------------------------------- System info ----
PCI slot                    : 0000:04:00.0
NUMA node                   : -1
[root@localhost centos]# nfb-dma
------------------------------ RX00 NDP controller ----
Received                    : 0
Discarded                   : 0

------------------------------ TX00 NDP controller ----
Sent                        : 0
```

Listing 6.8: Initializate `nfb` driver in guest VM, verify the mdev device using `nfb-info` and `nfb-dma` tools.

DMA transactions can now finally be tested, using the `ndp-generate` tool. Results are shown in listing 6.9 – the DMA do not work. The problem seems to be that the IOMMU in the host system is unable to find a valid translation between $GPA \rightarrow SPA$, `dmesg` in the host system shows that IOVA (Input-Output Virtual Address) does not correlate to PFN (Page Frame Number). The reason behind this might be questionable, the most logical explanation is that although the IOMMU has `nfb` device and mdev device in one IOMMU domain, it can not distinguish them internally (the IOMMU's granularity is too rough). To solve this problem, the PASID (auxiliary domains) technique (see 2.4, 5.2) with a finer IOMMU granularity needs to be used.

```
[root@localhost centos]# ndp-generate -i0 -p10 -s64
------------------------------- NDP generate stats ----
Packets                     :                    0
Bytes                       :                    0
Avg speed [Mpps]            :                (inf)
Avg speed L1 [Mb/s]         :                (inf)
Avg speed L2 [Mb/s]         :                (inf)
Time                        :                    0
[root@localhost centos]# nfb-dma
------------------------------ RX00 NDP controller ----
Received                    : 0
Discarded                   : 0

------------------------------ TX00 NDP controller ----
Sent                        : 0
cider(SL7) ~$ dmesg
[ 562.634421] Failed get IOVA for PFN 29204a0
```

Listing 6.9: Virtual machine setup with virsh tool.

6.2 Problems with the DMA, PASID experiments

With IOMMU's finer granularity, IOMMU's distinguishing capabilities improve. A design with a PASID is used in the test scenario but that's insufficient, in practice the IOMMU does not use the PASID automatically. To enable the IOMMU to work with PASID the `iommu_aux_()` family functions need to be invoked during the initialization part of the VFIO framework (as described in 5.2). The initialization of these functions should be automated,

based on the available hardware in the system. As have been shown, it all depends primarily on the CPU (IOMMU is part of the CPU). Evidence for this is the source code of the Intel IOMMU kernel driver, listing 5.3 or extract of it, listing 6.10.

```
if (!dev_is_pci(dev) || dmar_disabled ||
    !scalable_mode_support() || !iommu_pasid_support()) {
    return false;
}
```

Listing 6.10: Conditions for enabling auxiliary domains (PASID) in the IOMMU.

Unfortunately, not even the CPU in a testing scenario or any other Intel CPU server in Liberouter's organization has the capabilities for PASID and scalable mode, listing 6.11.

```
cider(SL7) ~$ cat /sys/class/iommu/dmar0/intel-iommu/ecap
f020df
```

Listing 6.11: Finding the capabilities of the IOMMU.

That is a major drawback, DMA simply can not be tested, finer granularity without PASID is unachievable. Perhaps more shocking is that, there are currently zero Intel Server (Xeon) CPUs on the market that support these capabilities, at the time of writing this thesis. Initial assumption was that it would be supported by the latest Intel® Xeon® Silver processor from the Intel® Scalable Processor series[1] (one of these CPUs from 2nd generation is available in the Liberouter server), but it turns out it is not supported. Only the next generation - 3rd generation Intel Scalable processors will support this, after confirmation from the Intel side (through mail contact with the support department).

So at the present, it's not possible to test PASID (thus DMA transfers) in the Intel CPUs. But what about the Intel rival, the AMD? As it happens we've got one Liberouter server with the newest Ryzen 3960X CPU from AMD. AMD officialy does not support auxiliary domains - these features are unique to the kernel driver of Intel IOMMU. Yet PASID is supported by the AMD. Sadly, the AMD IOMMU driver does not provide any "user" API to use this PASID feature (driver functions and symbols are used internally, they are not exported for outside usage), it is possible to modify the AMD IOMMU driver, however that is not this project's objective. The AMD IOMMU driver on the other side includes `amd_iommu_v2.c`[2], an extension of IOMMU implementation. If we look closely at the source code, we could find interesting functions, for example the `amd_iommu_bind_pasid()`. To activate PASID, the qemu-kvm must handle the mediated device binding to a process (VM). To do this it is necessary to modify the qemu-kvm, which is out of the question (too complicated and not the goal of the thesis).

Nevertheless, let's try to make a hack which helps us to test some other way around the DMA transfers. Hacks that are using the `amd_get_domain()` function, shown in listing listing 6.1. Instead of allocating a new domain for the VFIO group, use the domain of the parent PCI device. To completely exploit this, the device IOMMU domain needs to be properly configured.

This action is risky and could cause system instability or unpredictable behaviour, we have basically switched off device isolation so the system might share the same domain with a mediated device and with the original parent device, it is definitely not recommended. We did this only for research purposes, in order to test whether the DMA transfers work or not.

[1] https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html
[2] https://elixir.bootlin.com/linux/latest/source/drivers/iommu/amd_iommu_v2.c

```c
static int vfio_iommu_type1_attach_group(void *iommu_data, struct iommu_group *iommu_grp) {
    // domain->domain = iommu_domain_alloc(bus);
    domain->domain = amd_get_domain(container_of(iommu_device, struct pci_dev, dev));
}
```

Listing 6.1: Modified VFIO IOMMU type1 driver with AMD domain hack.

```
cider(SL7) ~$ insmod ~xperes00/vfio-mdev-pci-sample-driver/drivers/iommu/amd_iommu_v2.ko
cider(SL7) ~$ dmesg
[967.817837] AMD-Vi: AMD IOMMUv2 driver by Joerg Roedel <jroedel@suse.de>
cider(SL7) ~$ rmmod vfio_iommu_type1
cider(SL7) ~$ insmod ~xperes00/vfio-mdev-pci-sample-driver/drivers/vfio/vfio_iommu_type1.ko
     allow_unsafe_interrupts=1

[root@localhost centos]# ndp-generate -i0 -p10 -s64
------------------------------ NDP generate stats ----
Packets                    :                   10
Bytes                      :                  640
Avg speed [Mpps]           :                    0.588
Avg speed L1 [Mb/s]        :                  414.118
Avg speed L2 [Mb/s]        :                  320.000
Time                       :                    0.000
[root@localhost centos]# nfb-dma
----------------------------- RX00 NDP controller ----
Received                   : 0
Discarded                  : 0

----------------------------- TX00 NDP controller ----
Sent                       : 10
```

Listing 6.12: Testing DMA transfers with "hack" in AMD based system.

It's a success (results shown in listing 6.12), the DMA is working – 10 packets were sent, but this solution isn't stable. The host system starts freezing once we shut down the VM, and works unpredictably, which is the cost of exploiting the IOMMU. Multiple mediated devices can also not be created, as the domain could only be attached to one parent device, and mediated device is not a parent device. This hack is not feasible on Intel based systems, because the Intel IOMMU API has different concepts of domain allocation. This means that we cannot fully measure the throughput, cause the problems connected to DMA.

Chapter 7

Conclusion

This thesis deals with the problem of I/O virtualization in networking and the ultimate goal was to implement working software drivers for I/O virtualization. The necessary theoretical background was presented from the core elements to the more complex technologies such as virtio, vhost, IOMMU, PASID, SR-IOV, VFIO and media devices. There are several ways to address the problem of virtualization, from the pure software point of I/O virtualization, through the hardware SR-IOV virtualization up to the current scalable I/O virtualization techniques which are entirely new and not fully deployed. After extensive research and familiarization with the technologies, two methods were selected to accomplish the desired goals.

The desired goals of the new drivers are to achieve the best possible **data throughput** (compared to performance in a non-virtualized environment) and adequate **flexibility** (manageability) of created virtual devices. The objectives have been fulfilled. The first method chosen for driver implementation is `npp_vhost`. The reason for this choice is simplicity, as this driver implementation is based on complete software emulation. This approach gives us a complete freedom to create virtual device interfaces (the flexibility aim), but emulation is performed at the software layer which slows down final performance. In the initial driver tests, it was found that the throughput was not ideal, and the work on the driver had been stopped.

The further development has been switched to the second concept, the `nfb_mdev` driver. This concept is inspired by the 3. generation I/O virtualization (scalable I/O virtualization) technologies such as VFIO and mediated devices. The driver is based on the `mtty` mediated device driver and is extended to work with the NDK platform. Driver is able to create a number of virtual interfaces from the original parent driver (in the current version, the default value for a new virtual device is one Rx/Tx channel, but only small code adjustments may modify behavior for more dynamic interface variation). The driver is working as expected, only one drawback is that the DMA transfers are not working. In short, the latest available hardware on the market is incapable of the `PASID` principle, which is a necessarity for finer IOMMU granularity. All of this is required to isolate the devices, which allow DMA transfers.

Future work

The driver `nfb_mdev` which has been created during this thesis is currently in the prototype phase. The main point is to demonstrate the functionality of emerging technologies and to improve I/O virtualization that will be widely used in the future. Another reason the

driver is not fully deployed and used by the users is that the systems are not yet ready for this, there is actually no hardware capable of PASID and scalable mode available for testing. There is a certain potential for improving the project, trying to improve the work with mediated devices, testing the `PASID` on the new Intel CPUs, and measure the real throughput between the device and VM, which is not possible at the moment (DMA problems).

Bibliography

[1] Adam, A. and Ilan, A. *Achieving network wirespeed in an open standard manner: introducing vDPA* [blog]. October, 2019. Available at: https://www.redhat.com/en/blog/achieving-network-wirespeed-open-standard-manner-introducing-vdpa.

[2] Baolu, L. *[PATCH v7 0/9] vfio/mdev: IOMMU aware mediated device* [online, mailing list]. Visited: 15.5.2020. Available at: https://lwn.net/Articles/780522/.

[3] Baolu, L. *[RFC PATCH v2 00/10] vfio/mdev: IOMMU aware mediated device* [online, mailing list]. Visited: 15.5.2020. Available at: https://lwn.net/Articles/763793/.

[4] Bugnion, E., Nieh, J. and Tsafrir, D. *Hardware and software support for virtualization*. 1st ed. Morgan & Claypool, 2017. ISBN 978-1627056939.

[5] Corbet, J., Rubini, A., Kroah Hartman, G. and Rubini, A. *Linux device drivers*. 3rd edth ed. Beijing ; Sebastopol, CA: O'Reilly, 2005. ISBN 9780596005900.

[6] Fang, Y. *VFIO Mediated Devices Introduction* [online]. Visited: 16.1.2020. Available at: https://kernelgo.org/vfio-mdev.html.

[7] Hummel, V. *Vysílání paketů na 100 Gb/s Ethernetu*. Brno, CZ, 2014. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/16029/.

[8] Hummel, V. *Framework pro hardwarovou akceleraci 400Gb sítí*. Brno, CZ, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/19303/.

[9] Intel®. *Achieving Fast, Scalable I/O for Virtualized Servers* [online]. December 2019. White Paper. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/scalable-i-o-virtualized-servers-paper.pdf.

[10] Intel®. *Intel® Virtualization Technology for Directed I/O* [specification]. June 2019. Available at: https://software.intel.com/content/dam/develop/public/us/en/documents/vt-directed-io-spec.pdf.

[11] Intel®. *Intel® Scalable I/O Virtualization* [specification]. Visited: 14.1.2020. Available at: https://software.intel.com/sites/default/files/managed/cc/0e/intel-scalable-io-virtualization-technical-specification.pdf.

[12] Intel®. *Virtual Machine Device Queues* [online, white paper]. Visited: 15.3.2020. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtual-machine-device-queues-paper.pdf.

[13] Jia, N. and Wankhede, K. *VFIO based driver for Mediated device, MDEV driver and Mediated device Core Driver - kernel v5.6.13* [online, source code]. Visited: 11.5.2020. Available at: https://elixir.bootlin.com/linux/latest/source/drivers/vfio/mdev.

[14] Jia, N. and Wankhede, K. *(Nvidia) VFIO Mediated devices - kernel.org* [online, documentation]. Visited: 16.1.2020. Available at: https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt.

[15] Jia, N. and Wankhede, K. *Mediated virtual PCI serial host device driver mtty - kernel v5.6.13* [online, source code]. Visited: 8.5.2020. Available at: https://elixir.bootlin.com/linux/latest/source/samples/vfio-mdev/mtty.c.

[16] Kegel, A., Blinzer, P., Basu, A. and Chan, M. *AMD - IOMMU tutorial* [presentation]. 2016. Available at: http://pages.cs.wisc.edu/~basu/isca_iommu_tutorial/IOMMU_TUTORIAL_ASPLOS_2016.pdf.

[17] Lyon, T. and Williamson, A. *VFIO core and VFIO IOMMU DMA mapping support for Type1 IOMMU - kernel v5.6.13* [online, source code]. Visited: 11.5.2020. Available at: https://elixir.bootlin.com/linux/latest/source/drivers/vfio.

[18] Macko, S. *Přehledná správa virtuálních strojů v projektu OVirt*. Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21183/.

[19] Martin, E. P. *Deep dive into Virtio-networking and vhost-net* [blog]. September, 2019. Available at: https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net.

[20] Martínek, T. *PCI, PCI-X, PCIe* [course NAV - Návrh externích adaptérů]. 2019. Available at: https://bit.ly/2RbNHRK.

[21] Oasis technical committee. *Virtual I/O Device (VIRTIO) Version 1.1* [online]. Visited: 11.1.2020. Available at: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html.

[22] Omang, K. *Software and hardware support for Network Virtualization* [presentation]. 2019. Available at: https://www.uio.no/studier/emner/matnat/ifi/IN5070/h19/timeplan/net_virt1_hand.pdf.

[23] OSDev.org. *Wiki - PCI* [online]. Visited: 9.1.2020. Available at: https://wiki.osdev.org/PCI.

[24] PCI-SIG®. *PCI Express® Base Specification Revision 4.0 Version 1.0* [specification]. Visited: 11.1.2020. Available at: https://owncloud.cesnet.cz/index.php/s/05gAHt1JOxxVNQf.

[25] PCI-SIG®. *Single Root I/O Virtualization and Sharing Specification Version 1.1* [specification]. Visited: 12.1.2020. Available at: https://composter.com.ua/documents/sr-iov1_1_20Jan10_cb.pdf.

[26] Perešíni, M. *Automatická konfigurace obslužných nástrojů pro FPGA firmware*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/19925/.

[27] Portnoy, M. *Virtualization Essentials*. 1st ed. Sybex Inc, 2012. ISBN 978-1-118-17671-9.

[28] Remeš, J. *Virtualizace vstupních a výstupních operací v počítačových sítích*. Brno, CZ, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/18396/.

[29] Roedel, J., Duran, L., Woodhouse, D., Raj, A. et al. *IOMMU [Intel,AMD] kernel driver - v5.6.13* [online, source code]. Visited: 12.5.2020. Available at: https://elixir.bootlin.com/linux/latest/source/drivers/iommu.

[30] Sim, P. *KVM performance optimization for ubuntu* [presentation]. 2013. Available at: https://www.slideshare.net/janghoonsim/kvm-performance-optimization-for-ubuntu.

[31] Solomon, R. *PCI Express I/O Virtualization Explained* [presentation]. 2010. Available at: https://www.snia.org/sites/default/orig/sdc_archives/2010_presentations/thursday/RichardSolomon_PCI_Express.pdf.

[32] Tian, K. *Hardware-Assisted Mediated Pass-Through with VFIO* [presentation]. 2019. Available at: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Hardware-Assisted-Mediated-Pass-Through-with-VFIO-Kevin-Tian-Intel.pdf.

[33] Tian, K. *Intel® Scalable I/O Virtualization - LinuxCON* [presentation]. June 27, 2018. Available at: https://schd.ws/hosted_files/lc32018/00/LC3-SIOV-final.pdf.

[34] Tsirkin, M. S. *Virtio-net server in host kernel - kernel v5.6.13* [online, source code]. Visited: 6.5.2020. Available at: https://elixir.bootlin.com/linux/latest/source/drivers/vhost/net.c.

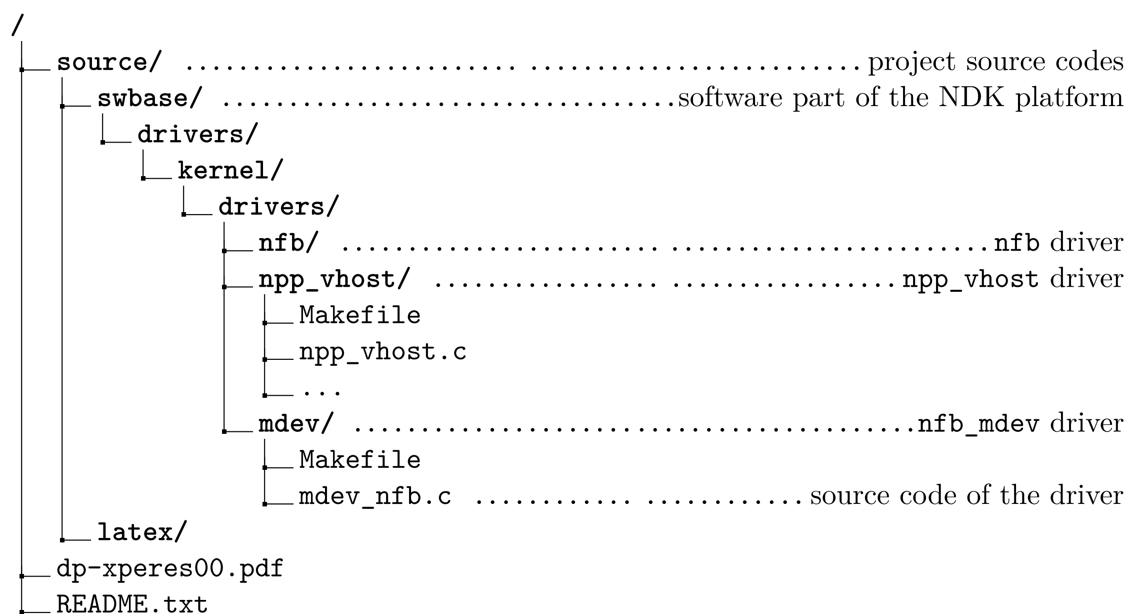
[35] Wang, J., Coquelin, M. and Tsirkin, M. S. *How deep does the vDPA rabbit hole go?* [blog]. October, 2019. Available at: https://www.redhat.com/en/blog/how-deep-does-vdpa-rabbit-hole-go.

[36] Wang, Z. *Intel GVT-g: From Production To Upstream* [presentation]. 2017. Available at: https://www.slideshare.net/xen_com_mgr/xppds17-intel-gvtg-from-production-to-upstream-zhi-wang-intel.

[37] Williamson, A. *VFIO - „Virtual Function I/O“* [online, documentation]. Visited: 10.5.2020. Available at: https://www.kernel.org/doc/Documentation/vfio.txt.

Appendix A

Storage Medium



Directory `source/` contains all source code files. The folder contains `npp_vhost` and `nfb_mdev` driver implementation. The directory structure matches the NDK software structure for better compilation integration (structure in 2.16). The attached files are not translatable separately, as the entire implementation took place in the private repositories of the CESNET research activity. Access to a private repository is required.

Directory `latex/` contains LaTeX source files for this Master's thesis. The folder also contains images that were used in the work.

File `dp-xperes00.pdf` is a PDF file containing the final version of the thesis text that was generated from the LaTeX source files.

File `README.txt` provides information about the directory structure of the attached storage media. The file contains details for the successful compilation of the software drivers that have been implemented in the thesis.

Appendix B

Handler of PCI extended capabilities in VHDL

This appendix contains the VHDL implementation source code for the PCI extended capability handler as part of NDK platform. This part - the firmware (VHDL) only implements the PCI configuration registers, not the functionality itself, which means that there must be another component to enable or disable PASID processing in PCI transactions for PASID operation.

```
-- pci_ext_cap.vhd: Extended capability handler for PCI
-- Copyright (C) 2017,2020 CESNET
-- Author(s): Martin Spinler <spinler@cesnet.cz>
--            Martin Peresini <xperes00@stud.fit.vutbr.cz>
-- SPDX-License-Identifier: BSD-3-Clause
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.dtb_pkg.all;

entity PCI_EXT_CAP is
generic (     -- Device Tree VSEC capability
    VSEC_BASE_ADDRESS       : integer := 16#480#;
    VSEC_NEXT_POINTER       : integer := 16#4A0#;
    CFG_EXT_READ_DV_HOTFIX  : boolean := true;
             -- PASID capability
    PASID_BASE_ADDRESS      : integer := 16#4A0#;
    PASID_NEXT_POINTER      : integer := 16#000#
);
port (        -- Interface description
    CLK                     : in  std_logic;
    CFG_EXT_READ            : in  std_logic;
    CFG_EXT_WRITE           : in  std_logic;
    CFG_EXT_REGISTER        : in  std_logic_vector(9 downto 0);
    CFG_EXT_FUNCTION        : in  std_logic_vector(7 downto 0);
    CFG_EXT_WRITE_DATA      : in  std_logic_vector(31 downto 0);
    CFG_EXT_WRITE_BE        : in  std_logic_vector(3 downto 0);
    CFG_EXT_READ_DATA       : out std_logic_vector(31 downto 0);
    CFG_EXT_READ_DV         : out std_logic
);
end entity;

architecture behavioral of PCI_EXT_CAP is
    constant VSEC_BASE_REG  : integer := VSEC_BASE_ADDRESS / 4;
    constant PASID_BASE_REG : integer := PASID_BASE_ADDRESS / 4;
    function dtb_words(data : in std_logic_vector) return integer is
    begin
```

```
39          return (data'length / 8 + 3) / 4;
40      end function;
41      function init_mem_32b (data : in std_logic_vector) return slv_array_t is
42          variable init : slv_array_t(0 to dtb_words(data)-1)(31 downto 0);
43      begin
44          for i in 0 to dtb_words(data)-1 loop
45              for j in 0 to 3 loop
46                  if (data'length / 8 > i*4+j) then
47                      init(i)(j*8+7 downto j*8) := data(i*32+j*8+7 downto i*32+j*8);
48                  else
49                      init(i)(j*8+7 downto j*8) := (others => '0');
50                  end if;
51              end loop;
52          end loop;
53          return init;
54      end function;
55
56      signal dtb_pf0 : slv_array_t(0 to dtb_words(DTB_PF0_DATA)-1)(31 downto 0) :=
        ↪  init_mem_32b(DTB_PF0_DATA);
57      signal dtb_vf0 : slv_array_t(0 to dtb_words(DTB_VF0_DATA)-1)(31 downto 0) :=
        ↪  init_mem_32b(DTB_VF0_DATA);
58
59      attribute ram_style     : string;
60      attribute ramstyle      : string; -- for Quartus
61      attribute ram_style of dtb_pf0  : signal is "block";
62      attribute ram_style of dtb_vf0  : signal is "block";
63      attribute ramstyle  of dtb_pf0  : signal is "M20K"; -- M20K, MLAB
64      attribute ramstyle  of dtb_vf0  : signal is "M20K"; -- M20K, MLAB
65
66      signal reg_dv           : std_logic := '0';
67      signal reg_dtb_pf0_addr : std_logic_vector(log2(dtb_words(DTB_PF0_DATA))-1 downto 0);
68      signal reg_dtb_vf0_addr : std_logic_vector(log2(dtb_words(DTB_VF0_DATA))-1 downto 0);
69      signal reg_dtb_pf0_data : std_logic_vector(31 downto 0);
70      signal reg_dtb_vf0_data : std_logic_vector(31 downto 0);
71      signal reg_cfg_ext_read_data : std_logic_vector(31 downto 0);
72      signal cfg_register     : integer;
73      signal cfg_function     : integer;
74      signal reg_pasid : std_logic_vector(31 downto 0); -- PASID register 'simulated'
75
76  begin
77     -- Address space:
78     -- 0x00: PCI-SIG spec: Vendor-Specific Extended Capability Header
79     --       Next capability Offset & Capability Version & PCI Express Extended Capability ID
80     --       0x000 & 0x1 & 0x000B
81     -- 0x04: PCI-SIG spec: Vendor-Specific Header: DTB capability
82     --       VSEC Length & VSEC Rev & VSEC ID
83     --       0x020 & 0x1 & 0x0D7B
84     -- 0x08: Allocated size for DTB ;0x0C: DTB real length
85     -- 0x10: DTB address register   ;0x14: DTB data register (RO)
86     -- 0x18: Reserved               ;0x1C: Reserved
87      cfg_register    <= to_integer(unsigned(CFG_EXT_REGISTER));
88      cfg_function    <= to_integer(unsigned(CFG_EXT_FUNCTION));
89
90      addr_dec: process(all)
91      begin
92          if (CFG_EXT_READ = '1') then
93              if    (cfg_register = VSEC_BASE_REG + 0) then
94                  CFG_EXT_READ_DATA <= std_logic_vector(to_unsigned(VSEC_NEXT_POINTER, 12)) & X"1" &
                    ↪  X"000B";
95              elsif (cfg_register = VSEC_BASE_REG + 1) then
96                  CFG_EXT_READ_DATA <= X"020" & X"1" & X"0D7B";
97              elsif (cfg_register = VSEC_BASE_REG + 2) then
98                  CFG_EXT_READ_DATA <= std_logic_vector(to_unsigned(DTB_PF0_DATA'length, 32)) when
                    ↪  cfg_function = 0 else
99                                       std_logic_vector(to_unsigned(DTB_VF0_DATA'length, 32));
100             elsif (cfg_register = VSEC_BASE_REG + 3) then
101                 CFG_EXT_READ_DATA <= std_logic_vector(to_unsigned(DTB_PF0_DATA'length, 32)) when
                    ↪  cfg_function = 0 else
```

```
102                                  std_logic_vector(to_unsigned(DTB_VF0_DATA'length, 32));
103             elsif (cfg_register = VSEC_BASE_REG + 4) then
104                 CFG_EXT_READ_DATA <= (31 downto log2(dtb_words(DTB_PF0_DATA)) => '0') &
                ↪   reg_dtb_pf0_addr when cfg_function = 0 else
105                                  (31 downto log2(dtb_words(DTB_VF0_DATA)) => '0') &
                                 ↪   reg_dtb_vf0_addr;
106             elsif (cfg_register = VSEC_BASE_REG + 5) then
107                 CFG_EXT_READ_DATA <= reg_dtb_pf0_data when cfg_function = 0 else reg_dtb_vf0_data;
108             elsif (cfg_register = VSEC_BASE_REG + 6) then
109                 CFG_EXT_READ_DATA <= (others => '0');
110             elsif (cfg_register = VSEC_BASE_REG + 7) then
111                 CFG_EXT_READ_DATA <= (others => '0');
112             -- PASID capability section
113             elsif (cfg_register = PASID_BASE_REG + 0) then
114                 CFG_EXT_READ_DATA <= std_logic_vector(to_unsigned(PASID_NEXT_POINTER, 12)) & X"1" &
                ↪   X"001B";
115             elsif (cfg_register = PASID_BASE_REG + 1) then
116                 -- PASID registers
117                 CFG_EXT_READ_DATA <= (10 => '1', 12 => '1', 16 => reg_pasid(16), others => '0');
118             else
119                 CFG_EXT_READ_DATA <= (others => '0');
120             end if;
121         else
122             CFG_EXT_READ_DATA <= reg_cfg_ext_read_data;
123         end if;
124     end process;
125
126     CFG_EXT_READ_DV <= reg_dv;
127
128     dtb_regp : process(CLK)
129     begin
130         if rising_edge(CLK) then
131             reg_dtb_pf0_data <= dtb_pf0(to_integer(unsigned(reg_dtb_pf0_addr)));
132             if (log2(dtb_words(DTB_VF0_DATA)) > 0) then
133                 reg_dtb_vf0_data <= dtb_vf0(to_integer(unsigned(reg_dtb_vf0_addr)));
134             else
135                 reg_dtb_vf0_data <= (others => '0');
136             end if;
137             reg_cfg_ext_read_data <= CFG_EXT_READ_DATA;
138
139             if (CFG_EXT_READ = '1' and reg_dv = '0' and (CFG_EXT_READ_DV_HOTFIX or (cfg_register >=
            ↪   VSEC_BASE_REG and cfg_register < VSEC_BASE_REG + 8) or (cfg_register >=
            ↪   PASID_BASE_REG and cfg_register < PASID_BASE_REG + 2))) then
140                 reg_dv <= '1';
141             else
142                 reg_dv <= '0';
143             end if;
144
145             if (CFG_EXT_WRITE = '1' and cfg_register = VSEC_BASE_REG + 4) then
146                 if (cfg_function = 0) then
147                     reg_dtb_pf0_addr    <= CFG_EXT_WRITE_DATA(log2(dtb_words(DTB_PF0_DATA))-1 downto
                    ↪   0);
148                 else
149                     reg_dtb_vf0_addr    <= CFG_EXT_WRITE_DATA(log2(dtb_words(DTB_VF0_DATA))-1 downto
                    ↪   0);
150                 end if;
151             end if;
152
153             if (CFG_EXT_WRITE = '1' and cfg_register = PASID_BASE_REG + 1) then
154                 reg_pasid <= CFG_EXT_WRITE_DATA(31 downto 0);
155             end if;
156         end if;
157     end process;
158 end architecture;
```

Listing B.1: VHDL source code of Device Tree VSEC capability and PASID capability support for PCI in the NDK platform.

Appendix C

Samples of `nfb_mdev` driver implementation, source code

```
static const struct mdev_parent_ops mdev_fops = {
    .create     = mtty_create,
    .remove     = mtty_remove,
    .open       = mtty_open,
    .release    = mtty_close,
    .read       = mtty_read,
    .write      = mtty_write,
    // ...
}
```

```
ssize_t mtty_read(struct mdev_device *mdev, char __user *buf, size_t count, loff_t *ppos){
    while (count) {
        size_t filled;
        if (count >= 4 && !(*ppos % 4)) {
            u32 val;
            ret =  mdev_access(mdev, (char *)&val, sizeof(val), *ppos, false);
        // ...
```

```
static ssize_t mdev_access(struct mdev_device *mdev, u8 *buf, size_t count,
                           loff_t pos, bool is_write){
    struct mdev_state *mdev_state;
    unsigned int index; loff_t offset; int ret = 0;
    // ...
    mdev_state = mdev_get_drvdata(mdev);
    if (!mdev_state) {
        pr_err("%s mdev_state not found\n", __func__);
        return -EINVAL;
    }
    // start working with mdev, mutex lock
    mutex_lock(&mdev_state->ops_lock);
    // calculate index and offset
    index = MTTY_VFIO_PCI_OFFSET_TO_INDEX(pos);
    offset = pos & MTTY_VFIO_PCI_OFFSET_MASK;
    // different access cases, PCI config space or BAR access, ...
    switch (index) {
    case VFIO_PCI_CONFIG_REGION_INDEX:
```

```
19          if (is_write) { // write request
20              handle_pci_cfg_write(mdev_state, offset, buf, count);
21          } else {         // read request
22              memcpy(buf, (mdev_state->vconfig + offset), count);
23          }
24          break;
25      case VFIO_PCI_BAR0_REGION_INDEX ... VFIO_PCI_BAR5_REGION_INDEX:
26      // ...
```

Listing C.1: Code snippets of the `mdev_parent_ops` structure, the `mtty_read()` function and the `mdev_access()` function inside `mtty` driver. Illustrates how guest VM read access to a mediated device is done.

```
1   struct vfio_iommu {
2       struct list_head    domain_list, iova_list;
3       struct vfio_domain  *external_domain;  // domain for external user
4       struct mutex        lock;
5       struct rb_root      dma_list;
6       unsigned int                dma_avail;
7       bool                v2;
8   }
9   struct vfio_domain {
10      struct iommu_domain *domain;  /* domain in generic IOMMU driver (Intel or AMD)
11                                       /drivers/iommu/iommu.c -> iommu/intel-iommu.c
12                                                              -> iommu/amd_iommu.c */
13      struct list_head    next, group_list;
14  }
```

```
1   static const struct vfio_iommu_driver_ops vfio_iommu_driver_ops_type1 = {
2       .name           = "vfio-iommu-type1",
3       .open           = vfio_iommu_type1_open,
4       .release        = vfio_iommu_type1_release,
5       .ioctl          = vfio_iommu_type1_ioctl,
6       .attach_group   = vfio_iommu_type1_attach_group,
7       .pin_pages      = vfio_iommu_type1_pin_pages,
8   };
```

```
1   static int vfio_iommu_type1_attach_group(void *iommu_data,
2                                           struct iommu_group *iommu_group) {
3       // Determine bus_type in order to allocate a domain
4       ret = iommu_group_for_each_dev(iommu_group, &bus, vfio_bus_type);
5       if (ret)
6           goto out_free;
7       // test for device/bus mdev or if it's a real pci bus
8       if (vfio_bus_is_mdev(bus)) {
9           struct device *iommu_device = NULL;
10          group->mdev_group = true;
11          // Determine the isolation type
12          ret = iommu_group_for_each_dev(iommu_group, &iommu_device,
13                                        vfio_mdev_iommu_device);
14          if (ret || !iommu_device) {
15              if (!iommu->external_domain) {
```

```
                INIT_LIST_HEAD(&domain->group_list);
                iommu->external_domain = domain;
            } else {
                kfree(domain);
            }
        }
        // set bus to real PCI bus (for IOMMU)
        bus = iommu_device->bus;
    }
    // allocate domain if doesn't exist
    domain->domain = iommu_domain_alloc(bus);
    if (!domain->domain) {
        ret = -EIO;
        goto out_free;
    }
    // attach domain to the device group
    ret = vfio_iommu_attach_group(domain, group);
}
```

```
static int vfio_iommu_attach_group(struct vfio_domain *domain,
                                   struct vfio_group *group) {
    if (group->mdev_group)
        return iommu_group_for_each_dev(group->iommu_group,
                                        domain->domain,
                                        vfio_mdev_attach_domain);
    else
        return iommu_attach_group(domain->domain, group->iommu_group);
}
```

Listing C.2: Code fragments of the VFIO_IOMMU_TYPE1 driver, shows the vfio and iommu key structures and their interactions.

Appendix D

Testing the implementation

```
[root@localhost centos]# dtc -q -I dtb -O dts /dev/nfb0
/dts-v1/;

/ {
        drivers {
                version = <0x10000>;
                ndp {
                        version = <0x1>;
                        rx_queues {
                                rx0 {
                                        mmap_base = <0x0 0x4000000>;
                                        mmap_size = <0x0 0x8000000>;
                                        size = <0x0 0x4000000>;
                                        numa = <0xffffffff>;
                                };
                        };
                        tx_queues {
                                tx0 {
                                        mmap_base = <0x0 0xc000000>;
                                        mmap_size = <0x0 0x8000000>;
                                        size = <0x0 0x4000000>;
                                        numa = <0xffffffff>;
                                };
                        };
                };
                mi {
                        mmap_size = <0x0 0x4000000>;
                        mmap_base = <0x0 0x0>;
                };
        };
        system {
                device {
                        card-id = <0x0>;
                        master {
                                numa-node = <0xffffffff>;
                                pci-slot = "0000:04:00.0";
                        };
                };
        };
        firmware {
                mi_bus {
                        compatible = "netcope,bus,mi";
```

```
43                      virtual {
44                              tx0 {
45                                      version = <0x10002>;
46                                      reg = <0x1200000 0x40>;
47                                      pcie = <0x0>;
48                                      compatible = "netcope,dma_ctrl_sze_tx";
49                              };
50                              rx0 {
51                                      version = <0x10002>;
52                                      reg = <0x1000000 0x40>;
53                                      pcie = <0x0>;
54                                      compatible = "netcope,dma_ctrl_sze_rx";
55                              };
56                      };
57              };
58      };
59 };
```

Listing D.1: Device tree (dts) printout of the **nfb** device as a mediated device inside the guest virtual machine.

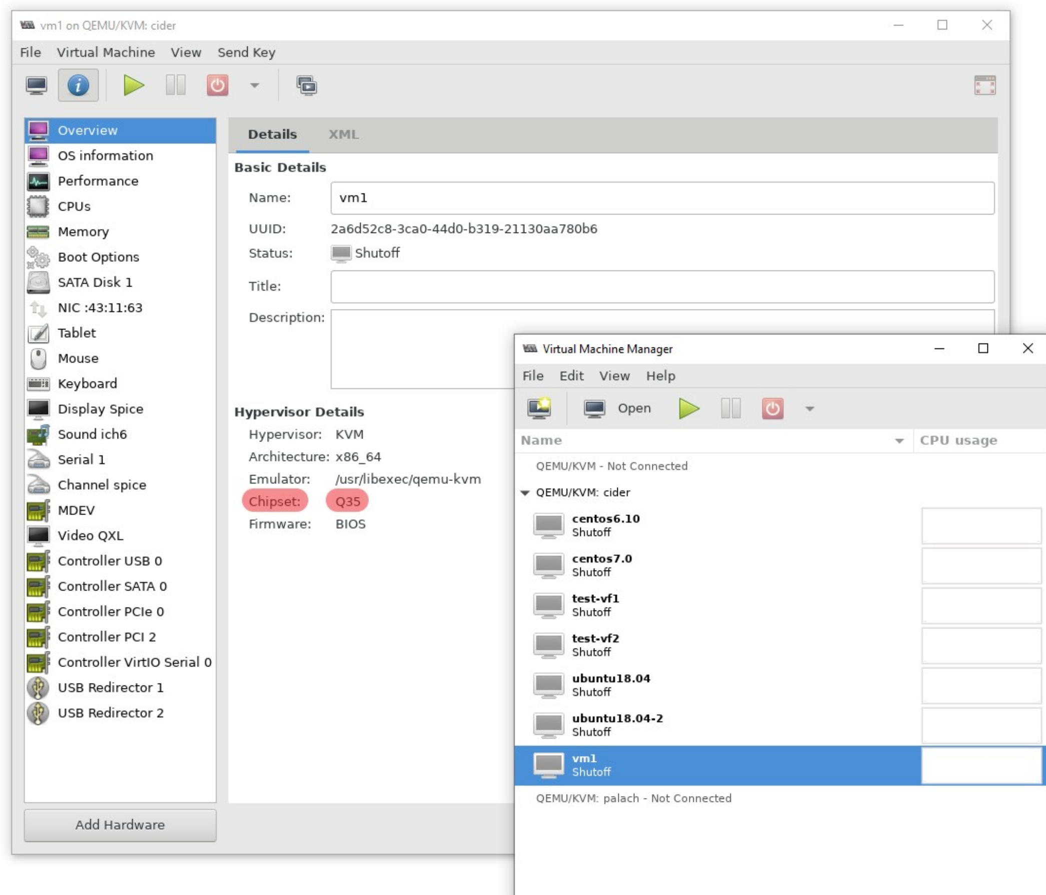


Figure D.1: Instance of a virtual machine setup using a virtual-manager graphical UI.

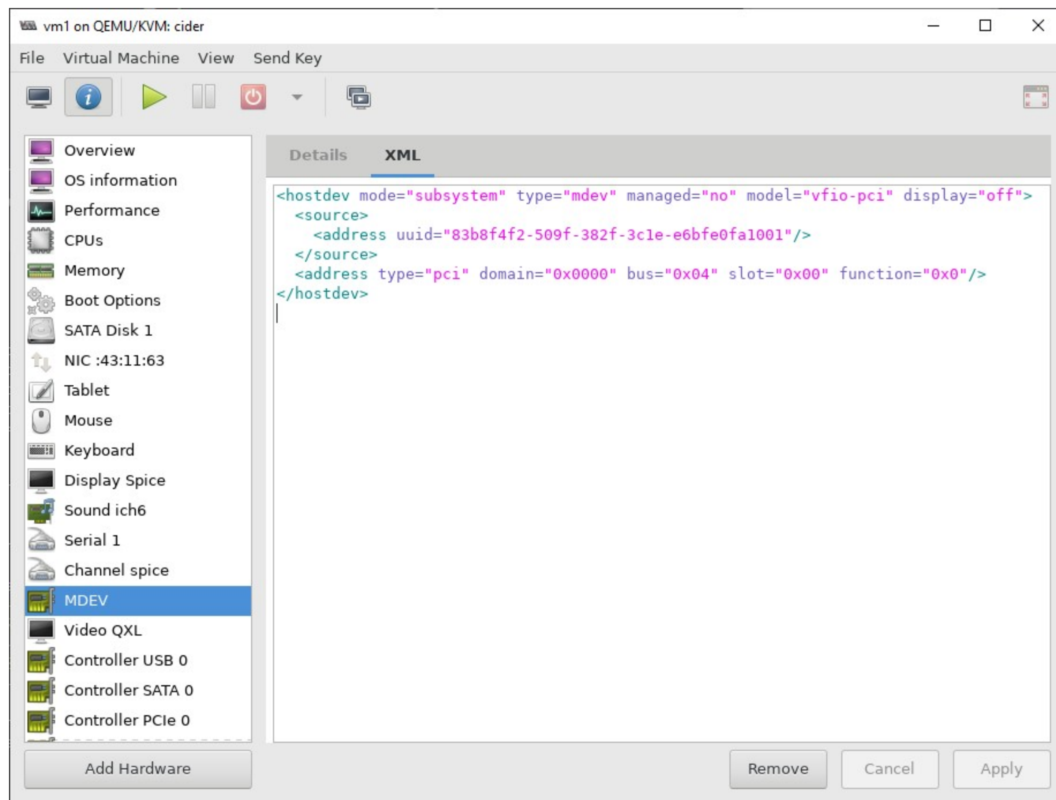


Figure D.2: Configuration of a mediated device in XML format inside a software virtual-manager UI.