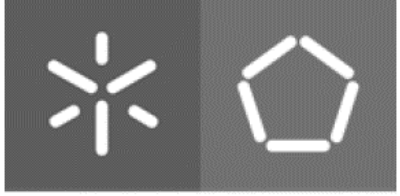
Universidade do Minho
Escola de Engenharia

João Rodrigo Lima Costa

VirtIO Infrastructure for a Static Partition Hypervisor: VirtIO-Net

dezembro de 2022



Universidade do Minho
Escola de Engenharia

João Rodrigo Lima Costa

VirtIO Infrastructure for a Static Partition Hypervisor: VirtIO-Net

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Pinto

dezembro de 2022

Agradecimentos

Esta dissertação de mestrado finaliza 5 anos de constantes "lutas" e não quero deixar de agradecer às pessoas que me ajudaram a supera-las. Seria impossível enumerar todos aqueles que me apoiaram durante todo este percurso, no entanto, queria exprimir às pessoas que mais diretamente influenciaram este sucesso.

Ao meu orientador, Dr. Sandro Pinto, por apresentar-me o conceito de virtualização que desde o início me cativou e por ter proposto um tema ambicioso. Ao meu coorientador, José Martins, que me acompanhou e me esclareceu durante toda a dissertação.

Aos meus colegas do Embedded Systems Research Group (ESRG), Francisco Rocha, Gonçalo Freitas e Nuno Capela, que trabalharam comigo no Bao, com quem partilhei o laboratório e com quem "discuti" diversas vezes.

Aos meus amigos, os Mirins, André Pereira, David Correia, Francisco Rocha, João Peixoto, Marcelo Amaral, Rafael Cachetas e Rui Esteves, que me acompanharam durante este 5 anos incríveis tanto nos melhores momentos como nos piores. Por estarem sempre lá, muito obrigado rapazes.

À minha família em especial, às mulheres da minha vida, a minha irmã, Matilde e a minha mãe, Lisete, Mãe, obrigado por todo o teu sacrifício para que eu pudesse atingir os meus objetivos. Obrigado por todos os conselhos e palavras amigas que me fizeram superar todos os desafios. Matilde, obrigado por apesar de irmã seres uma grande amiga.

Finalmente, um agradecimento especial a dois nomes já enumerados: Francisco Rocha e Matilde. Obrigado por ajudarem-me com o proof-read deste documento e obrigado por se mostrarem sempre disponíveis.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

O uso de sistemas embebidos tem crescido exponencialmente em indústrias como a automóvel ou aeronáutica. Isto tem levado a um aumento na complexidade dos sistemas, onde é necessário consolidar várias camadas de *software* com diferentes níveis de criticidade numa única plataforma de *hardware*.

Para aumentar a segurança destes sistemas, a indústria tem-se focado na tecnologia de virtualização, uma vez que a mesma permite a integração e o isolamento dos vários subsistemas. Recorrendo a um hipervisor é possível partilhar os recursos de *hardware* entre múltiplas máquinas virtuais (VMs). No entanto, os hipervisores tradicionais não foram desenhados para garantir os requisitos de tempo-real e de segurança. Por este motivo, hipervisores de particionamento estático, como o Jailhouse, que alocam os recursos de *hardware* estaticamente para as VMs em tempo de *design*, têm ganho cada vez mais protagonismo. Porém, o Jailhouse depende do Linux para iniciar e gerir as VMs, criando alguns problemas de tempo-real e segurança. Assim sendo, o nosso grupo de investigação focou-se em desenvolver o hipervisor Bao. O Bao implementa uma camada minimalista de *software* e não tem qualquer dependência de bibliotecas externas. A implementação atual do Bao dá acesso *pass-through* aos periféricos, não sendo possível a partilha de dispositivos.

O trabalho desenvolvido nesta dissertação consiste no desenvolvimento de uma infraestrutura que permite a partilha de dispositivos utilizando VirtIO. Esta infraestrutura deve ser genérica e deve ser compatível com as *front-ends* já existentes. A infraestrutura do VirtIO é implementada numa máquina virtual dedicada (*service guest*), cuja função é gerir as múltiplas virtqueues que permitem transmitir e receber dados de outras VMs que utilizam VirtIO. Ao contrário das soluções existentes, nesta dissertação, as *back-ends* do VirtIO são implementadas não no hipervisor, mas numa VM, resultando numa TCB reduzida para o sistema. A segunda parte da dissertação foca-se em implementar duas *drivers back-end*, i.e., uma para o VirtIO-console e outra para o VirtIO-net. A primeira *driver* é uma *driver* simples e é utilizada essencialmente para garantir a validação adequada da interface implementada. A segunda é mais complexa, mas é essencial para qualquer hipervisor moderno.

Abstract

The use of embedded systems has grown exponentially in industries such as the automotive or aeronautics. This led to an increase in the complexity of systems where it is necessary to consolidate several layers of software with different levels of criticality onto a single hardware platform.

To enhance the security of these systems, industry has been shifting towards virtualization, as the technology enables the safe integration and isolation of the various sub-systems. By leveraging a hypervisor it is possible to share hardware resources between multiple Virtual Machines (Virtual Machine (VM)s). However, traditional hypervisors were not designed to meet real-time and security requirements. For this reason, static partitioning hypervisors, such as Jailhouse, that statically allocate hardware resources to VMs at design time, have gained increasing attraction. However, Jailhouse depends on Linux to boot and manage VMs, which creates some issues for real-time, safety, and security. Under this light, our research group has designed and implemented Bao. Bao is a very thin layer of self-contained software, not having any external dependency. Bao's current implementation gives pass-through access to peripherals and device sharing is not possible.

The work developed in this dissertation is the implementation of an infrastructure that allows device sharing using Virtual Input Output (VirtIO). This infrastructure is generic and must be compatible with the existing front-ends. VirtIO's infrastructure is implemented in a dedicated virtual machine (service guest), whose aim is to manage the multiple virtqueues that allow transmitting and receiving data from the other VMs that use VirtIO. Unlike existing solutions, in our case, the VirtIO's back-ends are not implemented in the hypervisor but in a VM, resulting in a reduced Trusted Computing Base (TCB) for the overall system. The second part of the dissertation focuses on implementing two back-end drivers, i.e., one for VirtIO-console and another for VirtIO-net. The first driver is a simple driver, so it is used essentially to guarantee the proper validation of the implemented interface. The second one is more complex but is essential to any modern hypervisor.

Contents

List of Figures

List of Listings

Glossary

API Application Programming Interface

CPU Central Processing Unit

DMA Direct memory access

ESRG Embedded Systems Research Group

IO Input/Output

IP Internet Protocol

IPC Inter-Partition Communication

KVM Kernel-based Virtual Machine

MAC Media Access Control

MMIO Memory Mapped Input Output

OS Operating System

PMIO Port mapped Input Output

PV Para-virtualized

RAM Random Access Memory

RPC Remote Procedure Calls

RSS Receive Side Scaling

SLoC Source Lines of Code

SWaP-C Size, Weight, Power, and Cost

TCB Trusted Computing Base

TCP Transmission Control Protocol

UDP User Datagram Protocol

VirtIO Virtual Input Output

VM Virtual Machine

VMM Virtual Machine Monitor

1. Introduction

The use of embedded systems has grown exponentially in various industries such as the automobile and aeronautics. This has led to an increase in the complexity of systems, as multiple layers of software with different levels of criticality are required. On the other hand, there is an ongoing trend on the market to consolidate multiple subsystems in a single hardware platform, as it represents a reduction in Size, Weight, Power, and Cost (SWaP-C) [1].

By using multicore processors, it is possible to run multitask applications with different criticality levels at the same time in the same hardware platform. However, this can be dangerous as they can interfere with each other. Thus, one of the techniques used to give more security to these mixed-criticality systems, guaranteeing isolation of the subsystems, is virtualization [2].

Virtualization is a technique already established in cloud computing and it is getting increasingly used in embedded systems. It is a straightforward way to consolidate multiple software stacks and multiple operating systems onto the same hardware platform. Furthermore, it provides spatial and temporal isolation between operating systems [3, 4].

By using a hypervisor, it is possible to support several virtual machine guests sharing hardware resources. Nonetheless, traditional hypervisors, such as Kernel-based Virtual Machine (KVM) and Xen, are not prepared to guarantee some real-time or security system requirements. These hypervisors have a wide TCB because they depend on privileged running guests such as Linux. Furthermore, they have a penalty in performance, especially in the Input/Output (IO) access. Consequently, embedded hypervisors, such as XVisor or ACRN emerged. These hypervisors present reasonable TCBs and provide some sort of soft real-time guarantees [2, 5].

Since this wasn't a perfect fix, new disruptive solutions appeared: microkernel-based hypervisors and static partitioning hypervisors. These hypervisors guarantee real-time requirements and offer a smaller TCB which is praiseworthy since it leads to a more reliable hypervisor.

Jailhouse is an example of a static partitioning hypervisor. In these hypervisors, hardware resources are statically partitioned and allocated at the time of instantiating the virtual machines. However, Jailhouse relies on Linux to boot the system as well as to manage the virtual machines, which leads to an increase in the boot-time.

Thereby emerged Bao. Bao is an in-house implemented, open-source, lightweight hypervisor that provides a minimalist implementation from scratch without any external dependencies, which aims to

isolate mixed-criticality systems [1, 6]. Its main goal is to consolidate multiple software stacks and multiple Operating Systems onto the same hardware platform with strong isolation between VMs guarantying real-time requirements preserving a minimal TCB.

When it comes to peripherals, the current implementation of Bao gives pass-through access to existing devices on the platform, with no means to share peripherals between guests. In other words, if two guests need to use the same type of peripheral, there must be two devices, one for each guest. Emulation of devices and para-virtualization are two alternatives to pass-through. However, sharing devices leads to a decrease in performance, especially when emulating devices. Figure 1.1 presents both pass-through and emulation models: on the left the pass-through model (which Bao currently implements) and on the right, it is shown the emulation model that allows the sharing of devices.

The objective of the dissertation is to implement an infrastructure that allows the sharing of devices, abstracting the way that their data is accessed. For this, VirtIO will be used.

VirtIO is a popular standard based on para-virtualization techniques to virtualize peripherals because it offers a straightforward, efficient, standard and extensible mechanism. Before VirtIO, each hypervisor implemented its own device emulation interface. Instead of a variety of device emulation interfaces, VirtIO offers a common front-end driver (VirtIO driver implemented in the guest operating system as a device driver) framework that standardizes device infrastructures and increases code reuse across different virtualization platforms.

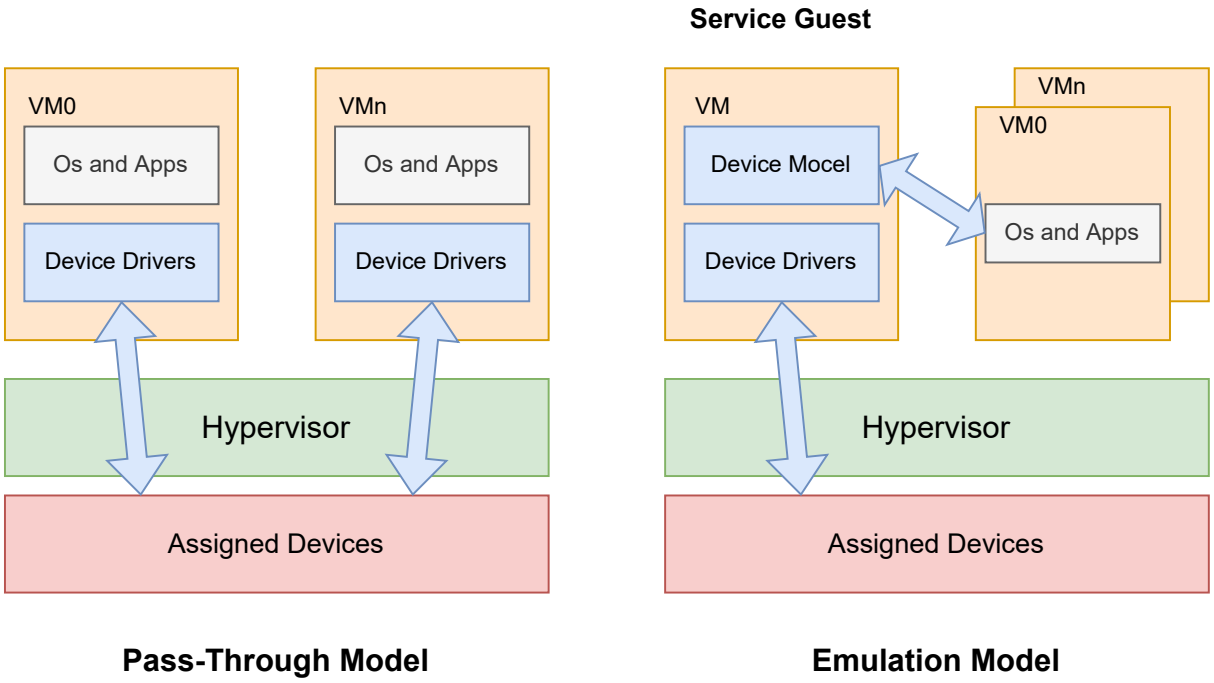


Figure 1.1: Pass-through vs Emulation of devices.

VirtIO's standard implementation was designed so that the back-end drivers are in the hypervisor. However, for static partition hypervisors such as Bao, this approach is not ideal since it is essential to keep

the kernel simple and minimalist. This way VirtIO's back-ends (drivers that are responsible for handling IO requests from the front-end driver and execute the requested operations in the physical devices) will be implemented in an unprivileged guest also called Service Guest as presented in the emulation model in Figure 1.1. The Service Guest is the only guest that can access to the physical devices by pass-through.

Nevertheless, there are some challenges to this implementation. For example, VirtIO has to be implemented in such a way that is compatible with existing front-ends. Moreover, the guest service will not have access to the memory of other guests because of the isolation between guests, unlike the standard implementation of the VirtIO whose hypervisor has access to the entire memory of all guests. For this reason, it will be necessary to use the hypervisor's Inter-Partition Communication (IPC) mechanisms to be possible to communicate between the VMs and the Service Guest.

1.1 Objectives

Looking at what was previously stated, a couple of objectives were formulated:

1. Implement the VirtIO infrastructure in Bao taking into account its requirements and limitations. It is essential to implement a generic infrastructure to be simpler to add new device drivers in the future;

2. Implement a device driver in the Service Guest to support VirtIO-console/VirtIO-serial to test the VirtIO infrastructure and guarantee its proper functioning;

3. Implement a driver for a network card mostly known as VirtIO-network so that it will be possible to communicate with an Ethernet card using VirtIO.

1.2 Document Structure

The remainder of this dissertation is divided into 4 chapters. Chapter 2 presents some basic concepts of virtualization, as well as some hypervisors types and architectures. Then, it continues with some background essential for the upcoming work of this dissertation, starting with virtualization and how VirtIO is included in it as well as VirtIO's structure. The chapter ends with some related work which implements similar infrastructure in well-known hypervisors. Chapter 3 presents the analysis, design, implementation and tests of the various stages to implement the communication between the VirtIO's back-end and front-end drivers as well as the mechanism on Bao to allow it. The Chapter 4 exhibits the analysis, design, implementation and tests of both VirtIO-console and VirtIO-net back-end drivers. Finally, Chapter 5 provides a summary of the dissertation as well as the conclusions regarding its implementation. It also presents future work to improve the interface.

2. Background and State of the Art

This chapter presents the fundamentals of this dissertation's development, as well as some existing related work. Firstly, it is given a contextualization about virtualization focusing on the virtualization for embedded systems followed by an explanation of the VirtIO and the concepts behind it.

2.1 CPU architecture

The Central Processing Unit (CPU) architecture that will be used in this dissertation is the ARMv8. ARMv8-A is the most recent ARM architecture. ARMv8 is usually used to describe the architecture that include the 32-bit (AArch32) and 64-bit (AArch64) execution states.

This architecture exception model defines four exception levels that determine the level of privilege. The levels go from EL0 to EL3, where EL0 has the lowest software execution privilege and EL0 the highest. Figure 2.1 presents how the software that usually run in each exception level in the non-secure state. In the EL0 run the normal user applications, in the EL1 the operating systems, in the EL2 the hypervisor and in the EL3 the low-level firmware.

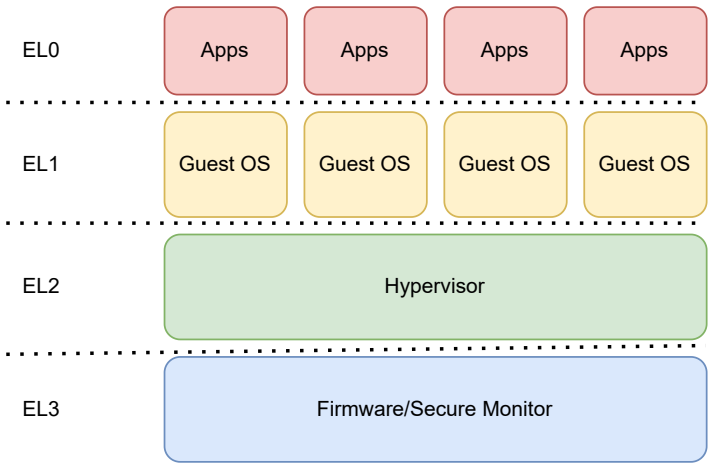


Figure 2.1: AArch64 exception levels.

ARMv8-A can also run in secure state, but this state does not support virtualization, meaning that it is not the interest area of this dissertation [7].

2.2 Virtualization

In computer systems, it is often needed a division into multiple abstraction layers, each of them with well-defined interfaces. The lower labels are usually implemented in hardware, while the higher ones are implemented in software.

Virtualization is a widely used term that can be applied to a substantial range of different technologies with different meanings in different contexts. Even though there is a lot of variance between applications, they have one quality in common: all of them use a set of physical hardware which acts as if there were multiple environments by dividing the hardware available into partitions.

Virtualization is an already well-established technique in cloud computing since it makes better use of multicore processors as it supports multiple Operating System (OS) environments on a single physical platform and provides the same isolation as separate physical servers, but with improved resource utilization [3]. This technique is game-changing because it allows multiple OSes to run on a single machine, being a straightforward way for consolidation, leading to a significant reduction in capital costs and the increase of energy efficiency. Additionally, it provides isolation and encapsulation from one virtual machine to another, improving security which is essential to highly secured applications as it increases system flexibility and brings a better fault tolerance [2, 8, 9].

Virtualization uses a single software layer that operates in hypervisor mode and is at least two orders of magnitude smaller than the usual OSes, thus it is likely to have fewer vulnerabilities [8]. The hypervisor or Virtual Machine Monitor (VMM), which is situated between the hardware layer and the OS, is the software layer that provides a software environment where programs or OSes can run just as if they were running on the hardware [8]. To provide this illusion of running in the hardware, the VMM has to provide an environment to the software that is identical to the original machine. It has complete control over the system's resources if it is a type-1 hypervisor (which will be explained in 2.2.5), but the programs running in this environment show a decrease in speed [8]. The basic software stack for system virtualization is represented in 2.2.

Although it has been studied since the 1960s [10], it was never necessary to use virtualization in embedded systems, because they used to be quite simple, single-purpose devices. Their main purpose was to achieve their goals with few resources. They usually had big constraints in what concerns the memory availability, the processing power or the battery charge. As a result, they regularly showed low to moderate software complexity. Beyond that, they had to guarantee real-time requirements, which is unusual in general-purpose devices [11].

Modern embedded systems, are embracing characteristics of general-purpose systems due to the needs of industries such as automotive or aeronautic. Their functionality is increasing and so is the software complexity. In the process, they have been trusted with sensitive data whose disclosure could cause serious damage to the system [12].

Since there is an ongoing trend to consolidate multiple subsystems into a single hardware platform

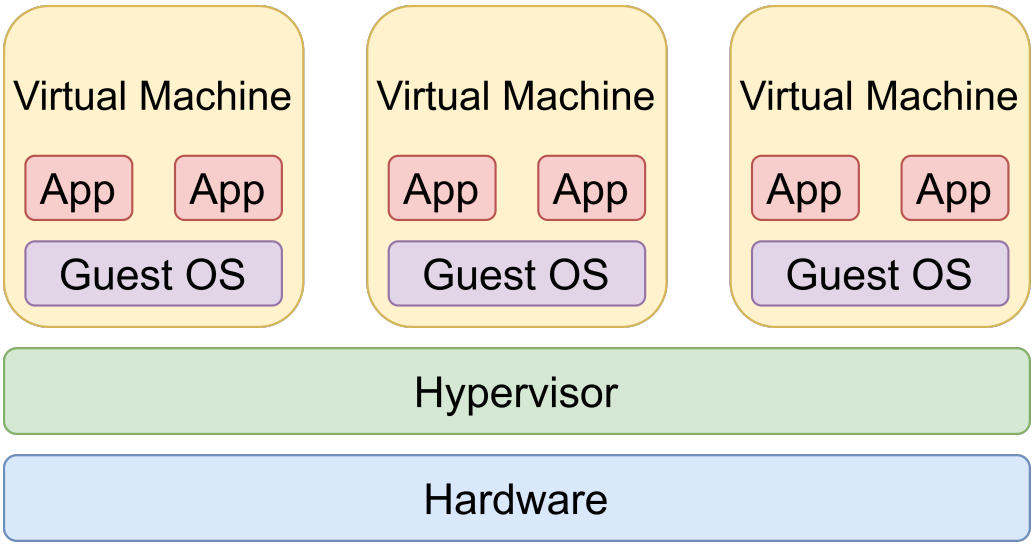


Figure 2.2: System virtualization stack.

in the referred industries, there is an urgent need to isolate both spatially and temporally the various subsystems because most of the systems have mixed-criticality. This means that different subsystems have different criticality levels and should not be capable of interfering with each other [12].

2.2.1 Full-Virtualization and Para-Virtualization

Virtualization can take different paths. Full-virtualization, also known as native virtualization, is a technique that does not require modifications to the guest OSes. It relies on the hypervisor to emulate the hardware. Full-virtualization allows well-known operating systems such as Linux or Windows to run inside the virtual machines without changes being needed. When the virtual board attempts to execute a privileged instruction, for instance IO request or to write in memory, it is caused a trap into the hypervisor, which emulates the hardware's behavior to the instruction and returns the results [8, 9].

As another option, para-virtualization can be used in order to replace the sensitive instruction with hypervisor calls (hypercalls) as present in Figure 2.3. These hypercalls trigger a trap in the hypervisor, which will process its parameters and provide the service desired by the guest. For this to happen, the hypervisor needs to define an interface composed of different system calls that can be used by the OS guests. It is also possible to remove all the sensitive instruction from the guest and force it to only use hypercalls [2, 8, 9]. Moreover, hypercalls provide more abstraction than emulation at the machine instruction level, thus, the use of para-virtualization brings a performance boost in comparison to full-virtualization [4].

It is proved that para-virtualization is advantageous in embedded systems as it provides the guests the knowledge that they are running in a VM. The access to device drivers or peripherals can be accomplished through hypercalls.

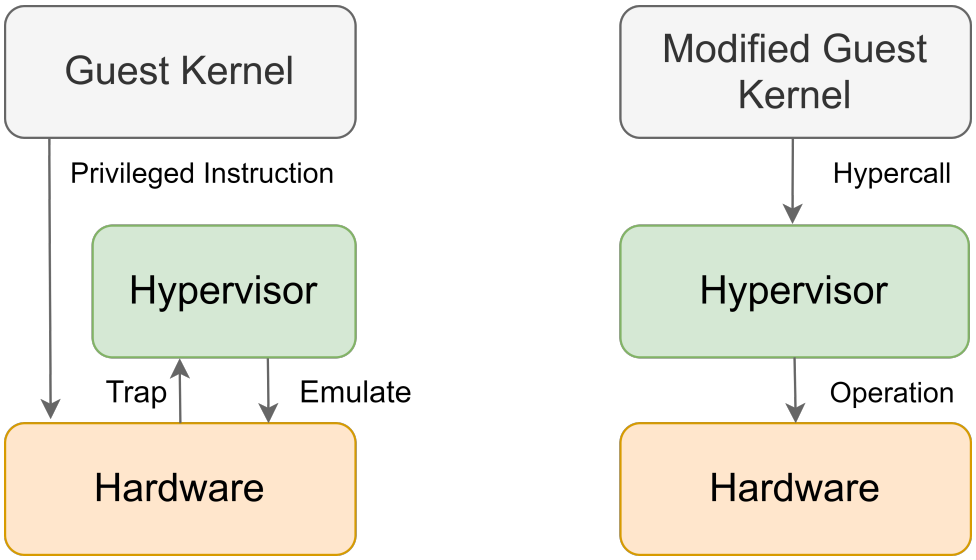


Figure 2.3: Full-virtualization vs Para-virtualization.

2.2.2 Trap-and-Emulate

Emulation provides VM portability and a wide range of hardware compatibility, which means the possibility of executing any virtual machine on any hardware, as the guest operating system interacts only with the emulated hardware. Emulation is the method by which the virtualization software duplicates the hardware component that is available to a virtual machine. The underlying physical hardware has no bearing on the emulated hardware.

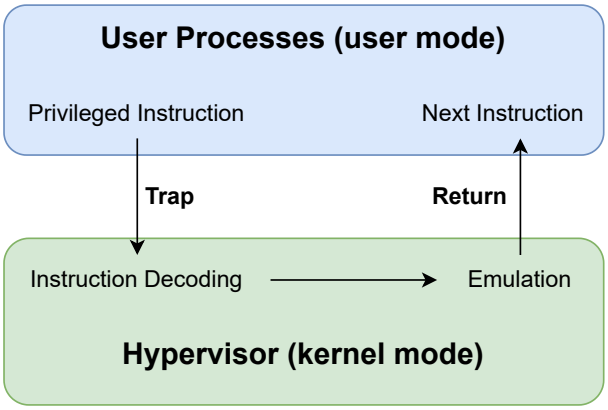


Figure 2.4: Trap and Emulate.

An approach known as trap and emulate uses the fundamentals of emulation while improving performance by employing interpretation selectively. The user programs and guest operating system of the virtual machines operate in a less privileged mode than the hypervisor, while the hypervisor runs in privileged mode. The software utilizes the hardware natively to execute the instructions. A trap to the kernel mode occurs when a privileged instruction is intended to be executed in virtual user mode, for instance.

If the platform being used provides virtualization extensions, the hypervisor can configure the action that leads to traps. This leads to a trap in the hypervisor and the execution is taken over by the hypervisor.

Figure 2.4, presents an example of how to trigger a trap. The hypervisor will interpret the instruction and emulate it. After the emulation, it returns to the user mode and proceeds to the following instruction.

2.2.3 Memory Virtualization

In a system that uses virtualization, each guest has its individual virtual memory tables. It is necessary to use address translation in each VM to translate the addresses of its virtual address space into addresses of the real memory. In a VM this real memory would correspond to the physical one, however, in system VM environment, the addresses of the VM's physical memory does not correspond to the real one. In fact, it is necessary an additional mapping to discover the address in the host hardware's physical memory.

The memory virtualization allows the combined total size of the real memory to be bigger than the physical memory of the system. This can only happen because the hypervisor maintains a swap space and it manages the physical memory by constantly swapping the guest's real pages into and out of its own swap space [13].

2.2.4 Peripheral Virtualization

When examining peripheral virtualization, firstly, it is necessary to settle if IO is Memory Mapped Input Output (MMIO) or Port mapped Input Output (PMIO). MMIO and PMIO are two methods used to perform IO between the CPU and the devices.

MMIO uses the same address space to address both memory and IO devices. Both memory and registers of the IO devices are mapped to address values, which means that when a CPU access an address, it can refer to a partition of the Random Access Memory (RAM) or an IO device. Therefore, the instructions used to access the memory can also be used to access devices. The addresses used by the CPU to access the devices must be reserved only for IO and must not be available for ordinary memory, otherwise, the data could be corrupted[1].

PMIO uses dedicated IO instructions designed specifically for performing IO. These must be privileged in order to achieve the resource control property of virtual machines. Using PMIO, IO devices have a different address space from memory, this is why sometimes PMIO is also refereed as isolated IO.

Peripheral virtualization can be assigned to one specific VM or shared amongst multiple VMs. The best solution as far as simplicity and efficiency are concerned is to assign them to one specific VM. This solution allows the VM pass-through to the peripheral register without any trap to the hypervisor, which means that the hypervisor does not have to handle IO requests.

[1] "Memory-Mapped I/O [MMIO]." https://embeddedartistry.com/fieldmanual-terms/memory-mapped-i-o/. Accessed: 2021–11-20.

The second solution permits the sharing of peripherals among multiple guests by emulating the peripheral in software. Virtual IO leads to a high overhead as all the accesses to the device must be trapped and its behavior emulated by software and it is also necessary to create virtual devices that will interact with the VMs.

To emulate a device, the MMIO area cannot be mapped by the hypervisor. When the guest tries to access this area, it is generated a trap. Then, the hypervisor identifies the address that was accessed as a virtual device and proceeds to emulate the device.

2.2.5 Hypervisors' Architectures

Going deeper on hypervisors, they can be divided into three topologies:

- **Type-1** hypervisors are also called native hypervisors or hardware-level virtualization because they run on a bare-metal or directly on the host's hardware. They have total control over the hardware and are able to access all the hardware resources of all guests OSes. Since all accesses to the hardware are managed by the hypervisor, the performance will depend only on the hypervisor itself. This conveys that type-1 hypervisors are the most suitable for systems that need to meet time constraints, such as embedded systems;

- **Type-2** hypervisors, also known as operating system-level virtualization, are hosted hypervisors which means that they are software applications running in a conventional OS, so they cannot control the hardware directly. That results in worse performance when set side by side with type-1 hypervisors as it is the OS's responsibility to access the hardware or execute any operation;

- **Type-1.5** hypervisors, are a compromise between both of the types presented. A part of the hypervisor runs directly on the hardware, while a privileged VM runs hosted extensions.

Type-2 hypervisors cannot be more secure than the general-purpose host OS where they are running. Ergo, they are not the best fit for critical applications in embedded systems.

Figure 2.5 presents all the topologies mentioned. On the left side, it is shown the type-1 hypervisor with direct access to the hardware, while on the right it is exhibited the type-2 hypervisor where the hypervisor is running in a host OS that has direct access to the hardware resources. Finally, below, it is presented the type-1.5 which is the mid-point between the other two types.

From here onward, the term "hypervisor" is used to refer to a type-1 hypervisor, since it is the topology used by the Bao hypervisor.

In security terms, the part of a system that can circumvent security policies and must be fully trusted is called the trusted computing base (TCB). In a system without memory protection, TCB is the complete system of potentially millions of lines of code. Such a large TCB might not be reliable. Traditional hypervisors have a vast TCB because they depend on privileged virtual machines such as Linux. They were designed without real-time constraints and have a big penalty in performance, especially in the IO access.

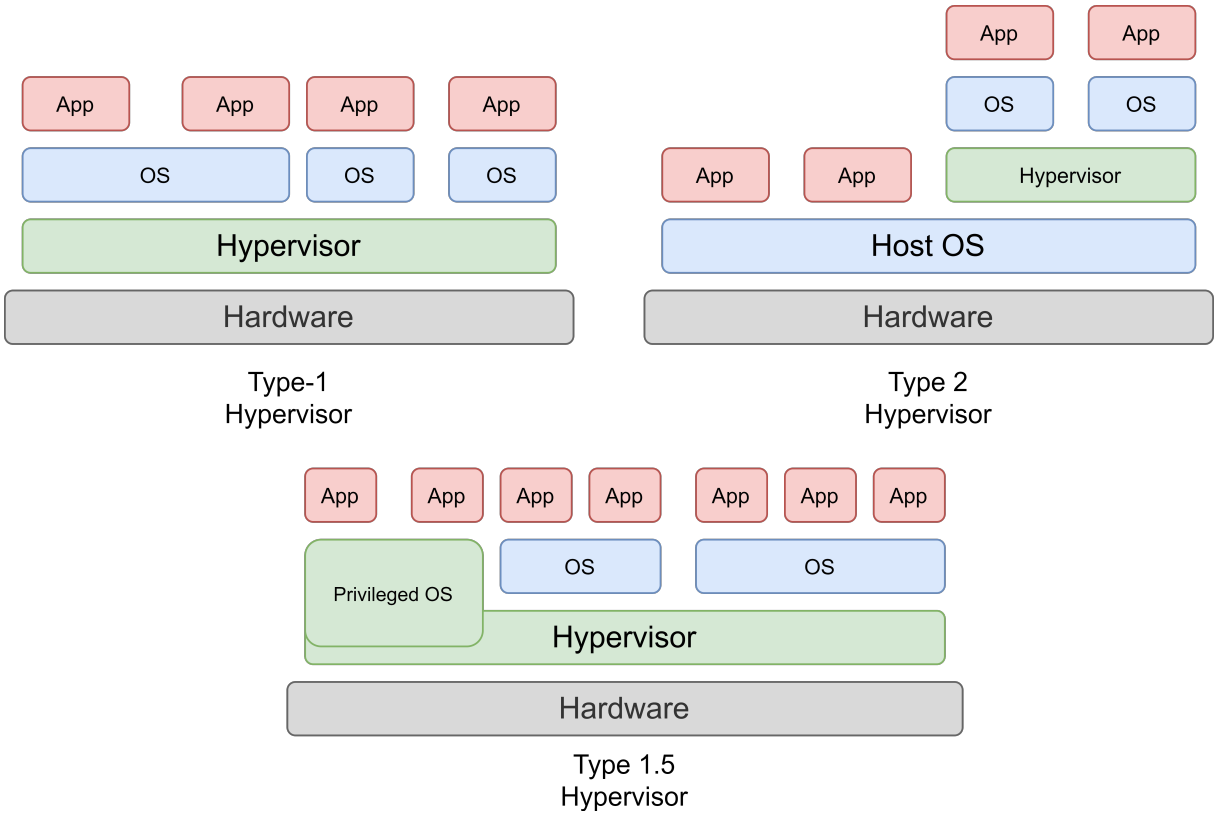


Figure 2.5: Types of Hypervisors.

A well-known example of this type of hypervisor is KVM. Microkernel based hypervisors surged to reduce the TCB.

Microkernel and Monolithic

Every hypervisor has to perform the same basic tasks: start and maintain the VMs, abstract system resources and share hardware resources. Even though they have the same purpose, their designs can diverge to meet specific requirements. Their distinctions can be summarized in two common hypervisor designs: microkernel and monolithic architectures.

Microkernel architectures were designed to respond to some problems due to the size of the kernel. A small kernel does not necessarily mean that the system is secure. However, security can be provided by isolating the services of the system. Microkernel architectures' hypervisor contains a few components using a separate partition to take care of functions such as storage, hypercalls, etc. Part of the TCB is inside the kernel space, as usual and the other part is outside the kernel space, which means it is located in the user space.

Monolithic architectures are the classical architectures used to supervise software. Unlike microkernel designs, monolithic architectures assemble all the subsystems. In monolithic designs, the hypervisor is entirely in the kernel space. This design does not use a separate partition and the VMs are directly above the hypervisor. Both, microkernel and monolithic architectures support hypercalls [8, 14].

The idea of microkernels emerged in the late '80s, but they were not very popular due to their poor performance. Recently, microkernel based systems have shown much better performance, yet not as good as the traditional (monolithic) systems [12].

Microkernel based hypervisors are usually lightweight microkernels that provide basic host hardware access and CPU virtualization. Some of these hypervisors run the device drivers under a driver virtual machine instead of under the common management VM. Examples of microkernel based hypervisors are OKL4 Microvisor and INTEGRITY Multivisor [2, 15].

2.2.6 Static Partitioning Hypervisors

Due to the need for hypervisors in environments with time constraints, there was a rise in the use of embedded hypervisors such as XVisor or ACRN [2, 16]. These hypervisors have a reasonable computing base (100K-10K Source Lines of Code (SLoC)) and it is possible to run multiple VMs on a physical CPU, but there has to be scheduling. They provide some sort of soft real-time guarantees.

The rise of industries like the automotive industry, dominated by mixed-criticality systems, where it is essential to guarantee real-time requirements, lead to the emergence of Static Partitioning Hypervisors. These hypervisors have a minuscule code base (5K-10K SLoC) which is ideal for critical systems. They also provide strong isolation and guarantee real-time requirements. However, static partitioning hypervisors have inefficient resource usage because the attribution of the hardware resources is static. Jailhouse and Bao are examples of these hypervisors.

2.2.7 Bao

Bao is a from-scratch implementation of a static partitioning hypervisor that targets mixed-criticality systems, so its main goal is to provide fault-containment and real-time behavior. This hypervisor does not have any external dependency except for standard platform management firmware.

Bao implements the static partitioning architecture (Figure 2.6) where the hardware resources are statically partitioned to the VMs. In Bao, the memory is allocated statically at initialization time, the IO of the virtual machines are pass-through only, the virtual interrupts are directly mapped to the physical interrupt and the virtual CPUs are assigned to the physical ones, following a 1:1 mapping. This means that each physical CPU runs a virtual CPU, which means that this hypervisor do not need a scheduler [1].

Usually, the VMs need to communicate with each other. Bao offers a mechanism of shared memory and asynchronous notifications triggered through hypercalls.

The three principles of Bao are:

- Minimality and Simplicity: The goal is to keep the code base as minimal and simple as possible. That is the reason why Bao is only implemented in architectures that provide hardware-assisted virtualization.

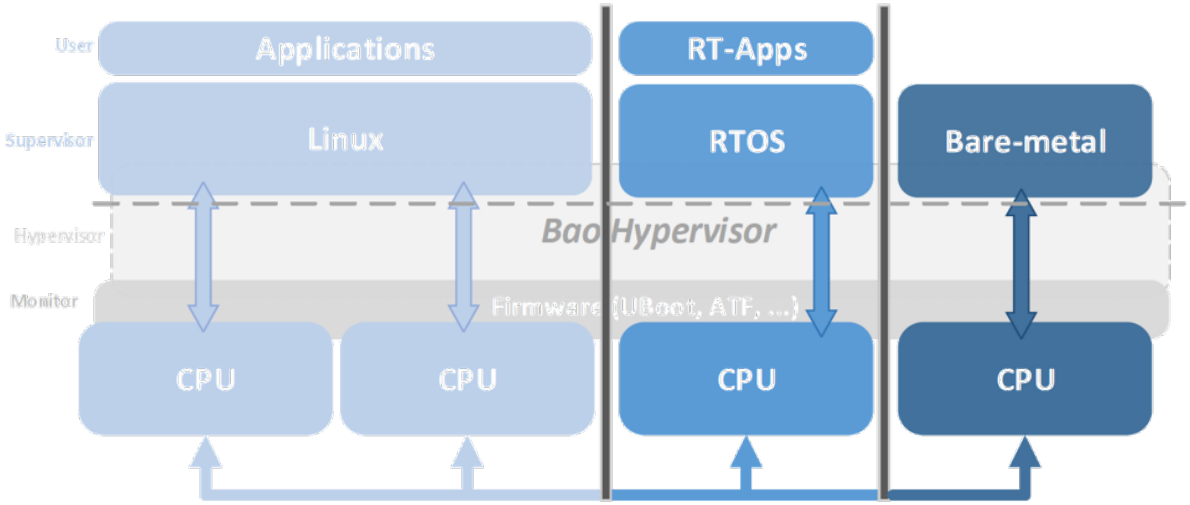


Figure 2.6: Bao Structure.

- Least Privilege: The goal is to ensure that every component in the system only has access to what is strictly necessary. Each core can access only the information that belongs to the VM assigned to it and the hypervisor cannot access directly the physical memory of the VMs.
- Thorough Isolation: Although the isolation provided by virtualization, the VMs still interact. For instance, the L2 cache is shared between the different VMs, which can lead to interferences. Bao uses a cache coloring mechanism to fight this issue. With this mechanism, the cache is divided by colors and it is possible to configure Bao in such a way that different VMs use different parts of the same cache, leading to better isolation but resulting in some problems, such as memory fragmentation [1].

Shared memory and notification via hypercall

As a static partitioning hypervisor, Bao implements an architecture where all the resources are statically partitioned and are exclusive to each VM. The memory is allocated at Bao's initialization, the IO is only possible to the guests by pass-through, the virtual interrupts and the virtual CPUs are assigned to physical ones. This means that each component has access only to what is essential. Each CPU has a private address space and only maps the physical pages it needs. This means that a CPU is not capable of accessing information of a VM that it does not run. However, they still need to communicate and for that reason Bao supplies simple primitives to communicate between VMs, based on a shared memory and asynchronous notifications that work as inter-VM interrupts, which are triggered using hypercalls. Besides, Bao is also not capable of directly accessing the physical memory of the VMs, meaning that when using hypercalls, the arguments must be passed not by reference, but by value using the processor registers [1, 6, 17].

Configuration

Bao allows the user to configure some parameters of the system. It allows to define the shared memories and theirs size, the amount of virtual machines that will be running, the devices that each VM has access, etc. These configurations are defined in the config.c file that will be used in Bao's compilation so that it can allocate everything statically.

The most fundamental parameter in what concerns this dissertation are the shared memory, the devices and the IPCs. The shared memory is configured as the following excerpt of code:

```
.shmemlist_size = 2,
.shmemlist = (struct shmem[]) {
    [0] = {.size = 0x00020000},
    [1] = {.size = 0x00020000}
},
```

Listing 2.1: Bao shared memory configuration.

The first parameter, *shmemlist_size* is the size of the shared memory list and defines the number of shared memory regions that can be used. After that, it is initialized the shared memory list, where it is possible to choose the shared memory's parameters. It is possible to choose the size, the memory coloring, the physical address, etc.

Inside each VM, it is possible to define how many devices it has access to. The following excerpt exhibits the configuration of a physical device:

```
.dev_num = 1,
.devs =  (struct dev_region[]) {
    {
        .pa = 0x09000000,
        .va = 0xFF010000,
        .size = 0x10000,
        .id = 0,
        .interrupt_num = 1,
        .interrupts = (uint64_t[]) {33}
    }
}
```

Listing 2.2: Bao device configuration.

Bao allows the configuration of the physical and virtual address of the device, as well as its size, the device ID and also the interrupt of the device.

Finally, the IPC configuration. Bao allows the use of shared memory regions, however it is necessary to define what are the virtual address that each VM must use to accessed them. To address this problem is necessary to configure a IPC:

```
.ipc_num = 1,
.ipcs = (struct ipc[]) {
    {
        .base = 0x70000000,
        .size = 0x00020000,
        .shmem_id = 0,
        .interrupt_num = 4,
        .interrupts = (uint64_t[]) {52, 54, 55, 56}
    }
}
```

Listing 2.3: Bao IPC configuration.

Here it is necessary to choose the virtual base address that will be used to access the shared memory, the size of the region, the shared memory ID that is going to be used as well as the interrupts associated to this shared memory. These interrupts will allow the communication between two isolated guests.

2.2.8 IPC

Usually, in computer science, IPC refers to inter-process communication which is a mechanism provided by the OS that allows a process to communicate with other processes and therefore the sharing of data between different processes. IPC mechanisms can differ according to the available resources. In microkernel-oriented OSes, the IPC is essential, since it is how the components that are outside the kernel interact via Remote Procedure Calls (RPC).

On systems that rely on virtualization, it is also necessary a communication mechanism. However, in this branch IPC refers to inter-partition communication or inter-VM communication instead of inter-process communication.

In traditional hypervisors, the inter-VM communication is performed by using Transmission Control Protocol (TCP)/Internet Protocol (IP) communication. Nevertheless, this method is not suitable for embedded systems hypervisors as the communication in these systems must be executed as fast as possible and emulating network controllers as well as going through multiple communication layers, induces significant overheads.

The IPC mechanism must be capable of guaranteeing separateness between VMs while they share data, otherwise, the system would lose one of the main goals of virtualization, isolation, which is essential to keep the system secure. A VM should not be capable of reading or corrupt VM's memory except for the exact communication buffers to which it has been granted access [18].

IPC Mechanisms

The IPC mechanisms used in virtualization systems can be classified regarding the synchronism, privilege level and overall message transfer mechanism. The IPC mechanism's data channel can be divided into:

- Shared Memory;
- Direct Transfer.

Shared Memory: This method consists of having a block of memory that can be accessed by multiple partitions, as long as it is given access to those partitions, which is a way to share data between them. This is an efficient method, however, it lacks security as it needs to allocate memory that can be accessed by more than a partition, which makes the system more vulnerable to attacks. To solve this problem, some strategies can be used, mainly by implementing an access control strategy supported by a kind of execution privilege level. Although, most systems implement their own shared memory IPC mechanism design, lately a great deal of academic work has emerged using the VirtIO's structure.

Direct Transfer: This method consists of copying the data directly between the transmitter and the receiver. This means that there is a dedicated hypervisor module responsible for message passing between VMs leading to performance overhead.

2.3 VirtIO

It is already well-known, as mentioned in Section 2.2.4, that virtualization increases the overheads to access IO peripherals such as network or storage devices. Thus, it is essential to reduce virtualization overheads since they are harmful especially for embedded systems. VirtIO can reduce the IO overheads in virtualized environments [19].

VirtIO is a virtualization abstraction Application Programming Interface (API). It was invented by Rusty Russell with the goal of creating a common layer for virtual devices such as Net, Block, Console and many others [20], for different hypervisors [21]. It emerged as an attempt to become the *de-facto* standard for virtual IO devices in para-virtualized hypervisors, because a standard means compatibility across different hypervisors and OSes [22].

Each VirtIO device uses two drivers: the front-end driver and the back-end. These two drivers are commonly called driver and device, respectively. The front-end driver is implemented in the guest operating system and works as a device driver. It replaces the device driver of a physical device. The back-end driver is implemented in the hypervisor or in a service guest. It is the back-end that handles the requests that come from the front-end driver. The back-end driver is responsible to execute all the requested operations in the physical devices, because it is the only driver access to physical devices. Figure 2.7 presents these two options. The left diagram presents the back-end driver in a dedicated guest whilst the right one places the back-end driver in the hypervisor.

The VirtIO para-virtualization mechanism is centered around data buffers, organized inside of a structure named virtqueue which encapsulates not only the shared data but also controls data. VirtIO devices can use multiple virtqueues to transmit and receive data (Tx virtqueue and Rx virtqueue for instance) [23]. Since the ring buffer structure used by VirtIO eliminates the need for mutual exclusion primitives or unnecessary copies, it is possible to achieve a big improvement in performance.

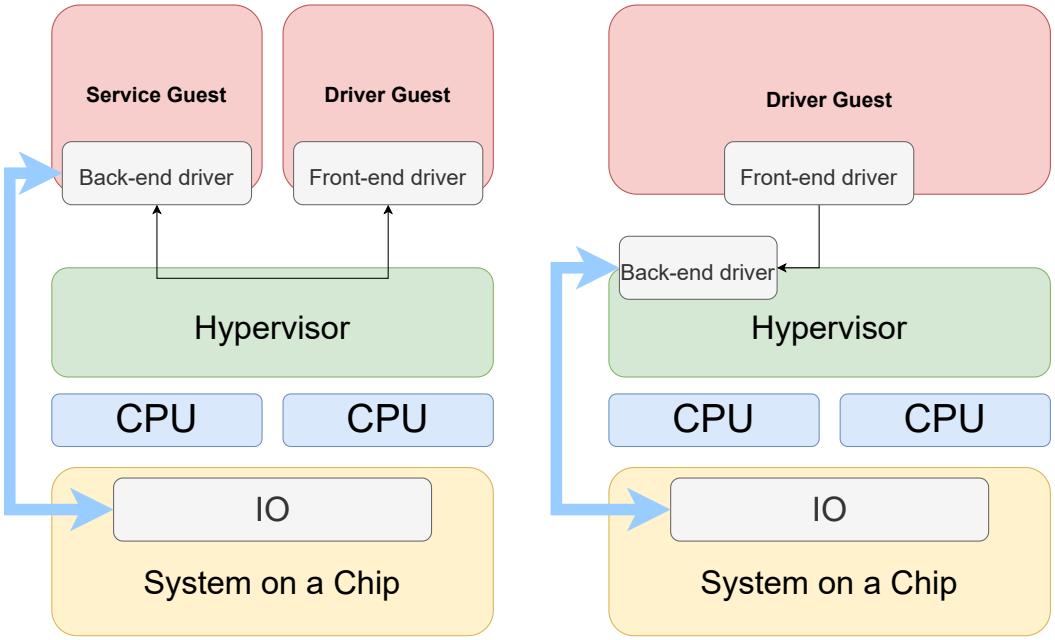


Figure 2.7: VirtIO back-end and front-end drivers.

To use VirtIO it is necessary to follow the specification [24], maintained by OASIS, which presents the structure as well as all the requirements of the implementation. The VirtIO interface consists of the following required parts:

- Device status field;
- Feature bits;
- Notifications;
- One or more virtqueues.

2.3.1 Device status field

The device status field supplies a low-level indication of the completed steps of the sequence necessary for the driver to initialize a device. The driver sets different bits to let the device know if it was recognized and if it is compatible. At the end of the feature negotiation, which will be explained in Section 2.3.2, the device can also set bits to let the driver know if all the features were accepted or if there was any problem in the negotiation [24, 25].

2.3.2 Feature bits

As mentioned before, after the initialization of a device, it is necessary to negotiate the features that will be used. This phase is vital in what concerns compatibility, since it defines the features that will be used.

Each VirtIO device offers all the features that it supports. Afterward, the driver reads the features and informs the device what is the subset that it accepts. If both sides agree, the driver will allocate and inform about the virtqueues to the device as well as all the configurations needed. This negotiation allows more compatibility because if the device has a new feature, the older drivers will not accept that feature. The same way around, if the driver has a feature that the device does not support, then the device will not use that feature since it was not offered [24, 25].

2.3.3 Notifications

The use of notifications has an essential role on VirtIO. There are three types of notifications:

- Configuration change notification;
- Available buffer notification;
- Used buffer notification.

Both configuration change notification and used buffer notification are sent by the device and received by the driver. A configuration change notification means that some configuration parameters have changed. A used buffer notification is utilized by the device to notify the driver that a buffer was processed and it became used. An available buffer notification is sent by the driver and received by the device and indicates that a buffer was made available on the virtqueue [24].

2.3.4 Virtqueues

The mechanism to transfer data on VirtIO is called virtqueue [26]. The virtqueue is one of the most important parts of the VirtIO since it contains all the information about the data that will be transported. As mentioned before, each device can have zero or more virtqueues, for instance, a simple network device needs at least two virtqueues, one to transmit and another to receive data.

To make a request available to the device, the driver must add an available buffer to the virtqueue. After that, it must send a notification informing the device that there is a new request. Afterward, the device processes and executes the request and once this is completed, it marks the buffer as used. Subsequently, the device sends a notification to the driver to notify it that the buffer has already been processed [24].

Each virtqueue can be compounded of up to three parts:

- Descriptor Area: used for describing the buffers;
- Driver Area: extra data supplied to the device by the driver;
- Driver Area: extra data supplied to the driver by the device.

Split Virtqueue

Until version 1.0 of the VirtIO's specification, there was only a type of virtqueues, the split virtqueue. The split virtqueue is based on a buffer descriptor table and two ring buffers, one for the device and the other for the driver. This format divides the virtqueue into three areas, where each area is writable by either the driver or the device, but not both:

- Descriptor Table – occupies the Descriptor Area;
- Available Ring – also called available virtqueue, occupies the Driver Area;
- Used Ring - also called used virtqueue, occupies the Device Area.

The descriptor table contains information about the data that will be transported from the device to the driver and vice versa. Each descriptor include the physical address of where the data is, as well as, the length. Moreover, each descriptor contains a set of flags that give more information about it. Each descriptor describes a data buffer that can be read-only or write-only for the device.

The driver uses the available ring to place the descriptor indexes that the device is going to consume. The parameters of the available ring can only be written by the driver. On the other hand, the device uses the used ring to return the used buffers to the driver. Unlike the available ring, the used ring can only be written by the device.

However, split virtqueues have some issues. Their structure lead to bad cache utilization since there can be several cache misses per request, which leads to a bad performance. This happens because the available and used rings use memory in a sparse way, which puts huge pressure on the cache usage [24, 27].

Packed Virtqueue

On version 1.1 of the specification it was presented an alternative compact virtqueue layout, called packed virtqueue, which redresses the bad cache utilization issue reducing, substantially, the overhead. Packed virtqueue solves this problem by merging the three rings (descriptor table, available ring and used ring) in a single location of the memory[2].

The use of this layout is negotiated by the VIRTIO_F_RING_PACKED feature bit.

Each packed virtqueue consists of three parts:

- Descriptor Ring – occupies the Descriptor Area;
- Driver Event Suppression – occupies the Driver Area;
- Device Event Suppression – occupies the Device Area.

[2] "[dpdk-dev] [PATCH v3 00/21] implement packed virtqueues." http://mails.dpdk.org/archives/ dev/2018-April/095470.html. Accessed: 2021–12-20.

Each descriptor of the descriptor ring consists of four parts:

- Buffer ID;
- Element Address;
- Element Length;
- Flags.

When the driver wants to send a request to the device, it writes at least a descriptor, describing the elements of the data that is going to be sent, into the descriptor ring. Then the driver sends a notification to the device. After that, the device will process the buffer and then it will write a used descriptor into the descriptor ring, by overwriting the descriptor sent by the driver and, afterward, it sends a notification back to the driver.

The descriptor ring is circular. The driver writes descriptors in order and when the end of the ring is reached it starts writing in the head of the ring. The device reads the descriptor in order, however, their processing can be completed out of order and for that reason, the device can write used descriptors out of order.

Besides the descriptor ring, the virtqueue is also composed of two event suppression structures, one for the device and the other for the driver. The device event-suppression data structure contains information to reduce the number of device events and it can only be changed by the device. The driver event-suppression data structure, contrariwise, contains information to reduce the number of driver events and can only be altered by the driver.

To replace the available and used rings used by the split virtqueue approach in packed virtqueue two wrap counters are used (one for the device and the other for the driver) as well as two flags inside each descriptor (VIRTQ_DESC_F_AVAIL and VIRTQ_DESC_F_USED). A descriptor is available if the bits VIRTQ_DESC_F_AVAIL and VIRTQ_DESC_F_USED are different and used if they are equal [24, 28].

Figure 2.8 presents the structure of a descriptor ring from a packed virtqueue. Note that the avail and used bits are not the only flags, there are also flags such as:

- next, that allows some sort of pointer to the next descriptor that should be processed, allowing the driver to supply a list of descriptors to the device;
- write, which allows the receiver to write data in the descriptor;
- indirect, which allows storing descriptors anywhere in memory, which is beneficial for large requests.

From now on, the term “virtqueue” is used to refer to a packed virtqueue, since it was the layout chosen for this dissertation’s implementation, taking into account that is the one that is more recent and that presents less issues.

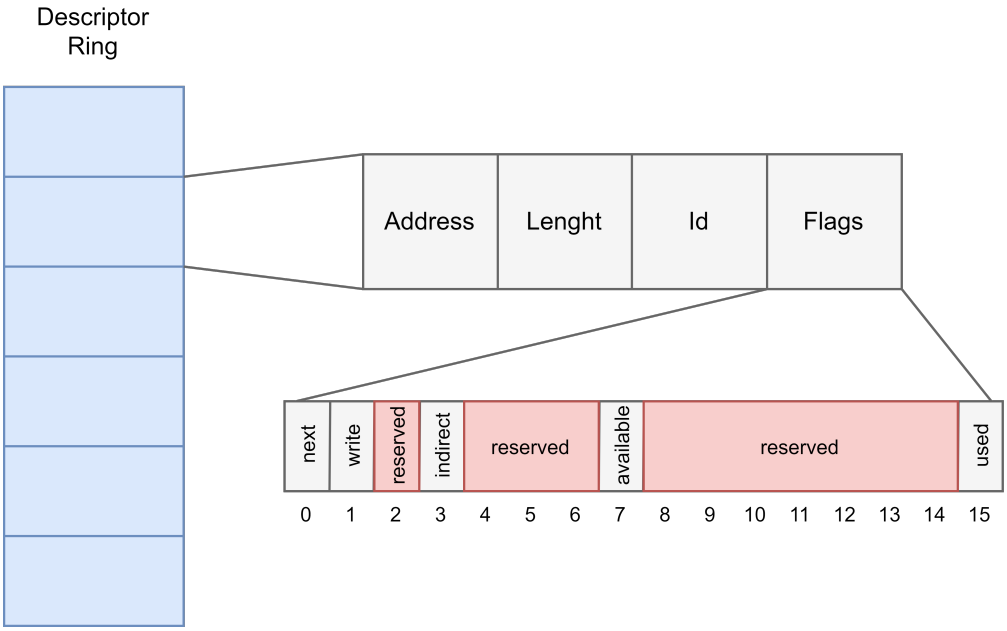


Figure 2.8: Packed Virtqueue.

2.3.5 Transport

Instead of creating a new bus from scratch, VirtIO devices are built on existing buses, which gives a straightforward way of communication between the front-end and the back-end drivers.

VirtIO can use various buses:

- PCI Bus;
- MMIO;
- Channel IO.

MMIO VirtIO devices provide 28 control registers followed by a configuration space. These control registers can be read or write-only by the front-end driver except the register *Status* that can be both written or read. This register is used in the initialization and returns the current device status flags and the register *QueueReady*, which is charge of notifying the device that it can execute requests from this virtual queue. The configuration space can also be written or read by the front-end.

There are 5 registers that are used to manipulate parameters related to the shared memory. These register will not be explained as they will not be used in this dissertation. The registers can be divided into seven groups:

- Registers used in the initialization of the device (*MagicValue*, *Version*, *DeviceId*, *VendorID* and *Status*);
- Registers used in the feature negotiation (*DeviceFeatures*, *DeviceFeaturesSel*, *DriverFeatures* and *DriverFeaturesSel*);

- Registers used to initialize and manipulate the virtqueues (*QueueSel*, *QueueNumMax*, *QueueNum*, *QueueReady* and *QueueNotify*);
- Registers that give the addresses of the virtqueues as well as the address of the device and driver areas (*QueueDescLow*, *QueueDescHigh*, *QueueDriverLow*, *QueueDriverHigh*, *QueueDeviceLow* and *QueueDeviceHigh*);
- Registers to manipulate the interrupts (*InterruptStatus* and *InterruptACK*);
- A register to guarantee the atomicity of the configuration (*ConfigGeneration*).

2.3.6 VirtIO-Console/VirtIO-Serial

The VirtIO console or serial device is a simple device for data transmission. A device can have one or more ports and each port have two virtqueues: one for input and the other for output.

Beyond the data virtqueues, for each device, there are also a pair of control virtqueues. These control virtqueues are used to transmit information between the device and the driver. They are capable of changing the negotiation between the driver and the device for establishing and terminating the data transmission. These two kinds of virtqueues have different implementations and different requirements. For instance, the data plane (data virtqueues) is required to move the packets quickly, while the control plane (control virtqueues) is required to be flexible in order to support different devices.

To transmit the data, one or more empty buffers are placed in the input virtqueue to receive the incoming data and the outgoing characters are placed in the output virtqueue to transmit data to the device [24].

2.3.7 VirtIO-Net

The VirtIO-net is a virtual ethernet card. As in VirtIO-serial, there are two layers: the control plane (control virtqueues) which is optional and the data plane (data virtqueues). Hence, the device uses at least two virtqueues for data transferring (one for transmission and one for receiving) [26, 29].

In the beginning, empty buffers are placed in the input virtqueue and the packets that are supposed to be transmitted to the device are placed in the output virtqueue in order. The third queue (control virtqueue) is used to control advanced filtering features [24].

The data is transferred in packets, which are made up of a header with a defined size and a data payload with a variable size. Nevertheless, because the receiver pre-allocated buffers for incoming traffic, the size of the pre-allocated buffers can be considerably smaller than the data payload buffer. The payload must be divided and copied into several receiver buffers in the communication infrastructure to account for this. Packets with huge payloads can be divided into many data transfers to make things simpler.

2.4 Related work

Most hypervisors use interfaces, such as VirtIO, to share devices or to communicate between VMs. This section presents some implementations related to this dissertation's work.

2.4.1 VM-to-VM communication

VirtIO was initially created in order to be possible to share devices on virtualized systems. However, there is some academic work that uses it as an interface to communicate between two different guests (inter-VM communication) since it is an efficient way of communicating. Several works [21, 23, 30] present implementations of this interface on different hypervisors.

The approach used in [23] is similar to Xen's Para-virtualized (PV) drivers, which will be explained further ahead, where a device is assigned by the hypervisor to a VM that acts as a multiplexer or broker for the other VMs that want to access the device.

2.4.2 KVM and Xen

KVM is an open-source hypervisor built into Linux. KVM uses QEMU, which is a hosted virtual machine emulator, to create virtual machines with virtual CPUs that the processor is aware of. When a special instruction, such as access to a device, is performed, the hypervisor informs QEMU of the cause of the pause so that the QEMU can perform the device emulation [31]. However, the device emulation leads to poor performance.

Xen is a Type-1.5 hypervisor, which provides full-virtualization to partition the host machine into different VMs, but it can also use para-virtualization [32]. Para-virtualization is used in Xen to reduce the cost and complexity of IO virtualization regarding device emulation. Xen uses a guest OS that runs a para-virtualized (PV) driver that works over an abstract device model that was exported to the guest. The real device can be either on the hypervisor or in a separate device driver domain that has privileged access to the device hardware [33]. This model is known as PV split driver model.

PV drivers are different, but still architecturally similar to VirtIO [32]. Just like VirtIO, they use IO descriptor circular rings. A ring is a queue of descriptors that are allocated inside a VM but can also be accessed by the hypervisor. These descriptors do not contain data, they only contain a reference to the memory place where the data is stored. The access to each ring is based on two pairs of producer-consumer pointers where a VM places the requests on a ring, increments a request producer counter, then Xen removes the requests to process them and increments the associated request consumer pointer. The responses are written on the ring, but this time the producer is Xen and the consumer is the VM [5].

When KVM emerged in the Linux scene, it did not have a paravirtual device model. The performance limitations of emulating devices were evident and adopting Xen's (PV drivers) approach was not appealing because it was implemented focusing on Xen and was not compatible with other hypervisors [26].

That is why KVM opted to use VirtIO. KVM supports IO para-virtualization using VirtIO [32]. KVM's VirtIO infrastructure is implemented in the hypervisor.

2.4.3 ACRN

ACRN is a type-1 hypervisor that was designed especially for embedded systems. This hypervisor uses the VirtIO specification to virtualize devices.

It uses a service OS that contains the back-end drivers as well as some other key components. The front-end drivers are located in the user OS that needs access to the devices. The communication between the back-end and the front-end drivers is done using the virtqueues that are placed in shared memory so that both drivers are able to access them. The virtqueue layout used in this hypervisor's is the split virtqueue [16].

One of the peculiarities of this hypervisor's VirtIO implementation is that it disposes of two framework implementations:

- Back-end service in user-land (suitable for devices that do not have performance requirements);
- Back-end service in kernel-land (suitable for performance-critical devices).

The back-end drivers are different in these two frameworks, but the front-end drivers are the same for both of them.

2.5 Conclusions

Besides the fact that the hypervisor under which the work of this dissertation is going to be implemented (Bao) does not have an interface to share devices, the implementation is going to follow different paths from the implementations presented.

Section 2.4 presented some academic work using VirtIO not to virtualize devices but to run as a VM-to-VM communication interface. This is an interesting feature for hypervisors that do not possess a way to communicate between OS guests. Even though there are some differences due to Bao's infrastructure, this approach will be used in the initial phase of the implementation since it is a simple way to understand if the data transmitted using VirtIO is correct and if all the interface is working as it is supposed to.

KVM's initial approach does not fit the real-time requirements since the device emulation increases substantially the time needed to transfer data between the device and the driver. For this reason, this approach is not worth considering.

Xen's PV drivers are an interesting implementation. However, it was implemented for Xen and it has a considerable amount of problems as far as compatibility in other hypervisors is concerned. In fact, this is one of the reasons for the emergence of VirtIO.

The ACRN implementation of VirtIO has a lot in common with the implementation aimed at, in this dissertation. Besides the fact that the virtqueues are stored in a shared memory region, it uses a service guest that contains most of VirtIO's infrastructure. However, the big problem of this implementation is the use of split virtqueues. As mentioned in Section 2.3.4, this layout has some issues which lead to a bad performance. So, this dissertation aims to implement the virtqueues using the newest layout alternative (packed virtqueue).

Furthermore, no implementation of VirtIO in a static partitioning hypervisor was found where the interface is entirely in a service guest and where the descriptor rings are in a shared memory region.

3. VirtIO Interface

This chapter explains all the steps taken to implement the most essential part of this dissertation: the VirtIO interface. It is divided into three groups.

The first group presents the first step of the implementation, the creation of an interface similar to VirtIO but using hypercalls to send notifications between VMs. The main goal of this step is to prove that it is possible to communicate effectively between two guests by using virtqueues with descriptors that describe buffers of data.

The second section shows the VirtIO interface itself, where it is explained its functioning by presenting the design of the interface as well as the implementation.

The third section explains the mechanism created in Bao to make it compatible with VirtIO. It explains every step in which the hypervisor needs to take action and how it reacts to the access of MMIO registers that belong to VirtIO devices.

3.1 Inter-VM communication using virtqueues

The implementation phase of this dissertation was initiated with the creation of an infrastructure similar to VirtIO which main goal is to test the communication between two isolated VMs using virtqueues. However, this infrastructure uses shared memory to store the descriptors so that is possible to share this information between two virtual machines and uses the Bao's inter-process communication through hypercalls to notify the service guest. In this way, there is no need to modify Bao's source code, which means that it could be done in every hypervisor that allows communication between two virtual machines.

This implementation's goal is to enable sending messages from one virtual machine to another. Therefore, two virtual machines are used: the driver guest and the service guest. The driver guest is the driver that sends the first message. The service guest is the virtual machine that receives the message.

An essential part of this implementation is the usage of descriptors similar to the ones used in VirtIO. Despite using the same format, the flags of the descriptors that are mandatory in VirtIO are not used in this initial implementation. Each descriptor is 16 bytes where: the address occupies 8 bytes, the length of the data 4 bytes, the ID of the descriptor 2 bytes and finally the following 2 bytes are empty because they are used to store the flags of the descriptor.

This first approach starts with only one way of communication (from the driver guest to the service guest), as it is possible to notice in the Figure 3.1. Initially, a simple message is created. The driver guest puts it in a shared memory address and creates a descriptor with the information of the message (1). This descriptor is also stored in a shared memory address. This address is also known by the service guest. After that, the driver guest uses a hypercall to notify the service guest that there is a new message (2). This hypercall is used in such a way that triggers the interrupt that is associated with the shared memory that is storing the message. Then, the service guest receives the interrupt, reads the descriptor and accesses the message by using the descriptor's information (3). In the end, the service guest prints the data so that it is possible to know if the data was correctly sent from the driver guest to the service guest.

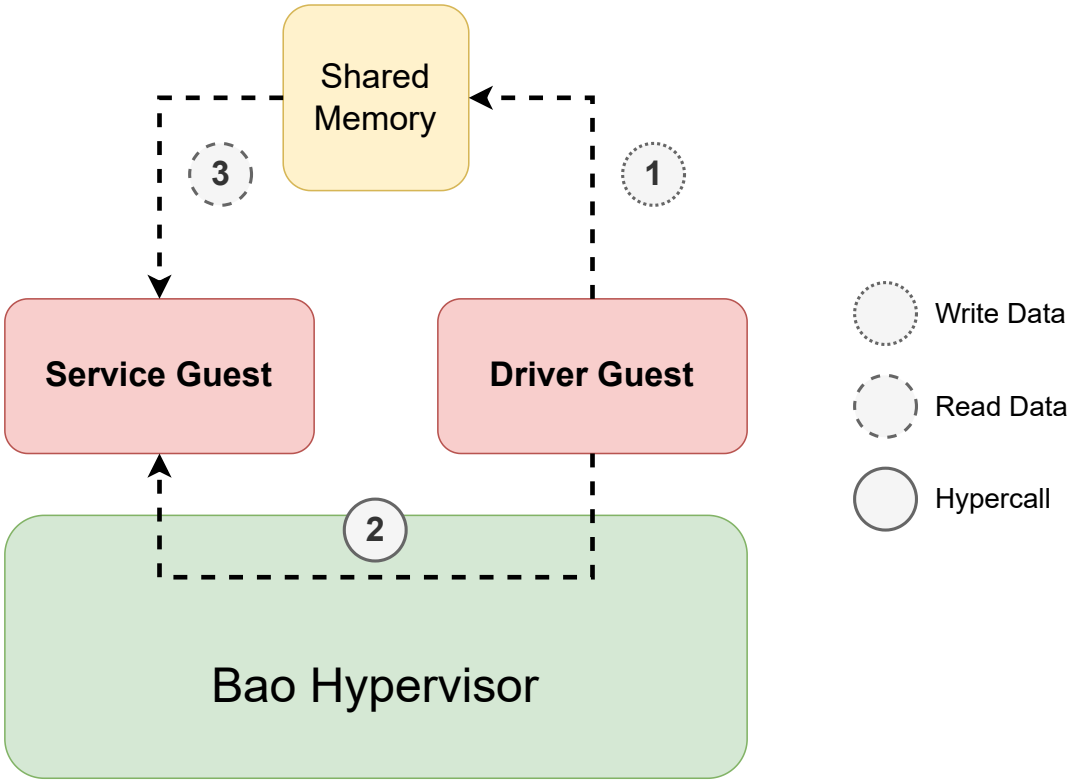


Figure 3.1: First Implementation with one communication way.

After this first approach, the mechanism was created so it would be possible to communicate both ways. This means that it is possible to send messages from the driver guest to the service guest and vice versa. Figure 3.2 presents the steps that this mechanism follows.

Every time there is a new message which needs to be sent, the driver guest creates a descriptor with the data's information and puts the descriptor as well as the message in the shared memory as in the first implementation (1). Thereon, it is used a hypercall to notify the service guest (2). The service guest receives an interrupt as a result of the hypercall and, in its callback, the descriptor is read (3). Following, the service guest prints the data sent by the driver guest by reading the descriptor's information and accessing the memory address where the message is. Then, the procedure is done in the opposite way. At the end of the interrupt callback, the service guest creates a new message to alert the driver guest that

the message was received. It places the message in the shared memory and not only creates but also stores a descriptor with the information of this message (4). Finally, the service guest uses a hypercall to notify the driver guest (5). The driver guest receives the interrupt, reads the descriptor created by the service guest and accesses the message (6). In the end, it prints the message to guarantee that everything has gone as planned.

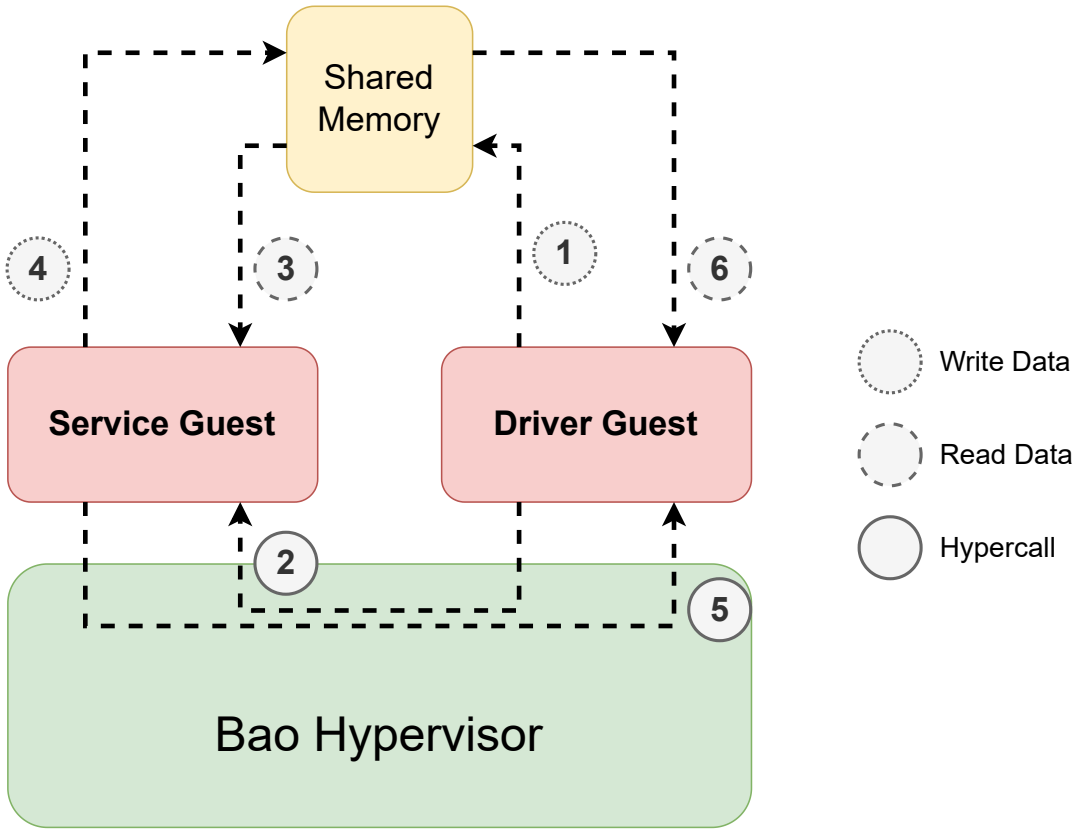


Figure 3.2: First Implementation with two ways.

Although being used only a single shared memory buffer for vm-to-vm communication using virtqueues, as this is a 1-to-1 interaction, in order to guarantee that more than one device can be served by the service guest, using a shared memory for each device is essential and consequently different callbacks for the devices in the service guest must be used. In this way, the service guest is able to understand which front-end driver generated the interrupt and act accordingly.

3.1.1 Implementation

Moving on to how the interface was implemented. Firstly, it was necessary to create a structure that contains the general information of each device:

```
1 struct virtio_device {
2     char *shmem_desc_base;
3     char *shmem_data_base;
```

```
    size_t shmem_desc_size;
    size_t shmem_data_size;
    long ipc_id;
    int vq_number;
    int *vq_interrupt_numbers;
    char data[50];
};
```

Listing 3.1: VirtIO device structure for inter-VM communication.

This structure contains the information of where the base address of the descriptors and of the data sections is, as well as their sizes, the ID of the shared memory that was used, the number of virtqueues that the device has and their IDs. This struct also contains an array that will store the data that it received.

This interface is similar to VirtIO's interface. Therefore, there is at least one virtqueue associated with each device. It is crucial to create a new struct that incorporates all the information of each virtqueue:

```
struct virtq {
    long used_wrap_count;
    long next_used;
    long avail_wrap_count;
    long next_avail;
    long size;
    char *shmem_queue_base;
    char *shmem_data_base;
    short id;
    /* Descriptors */
    struct pvirtq_desc *desc;
};
```

Listing 3.2: Virtqueue structure for inter-VM communication.

This struct contains two counters (*used_wrap_count* and *avail_wrap_count*) explained in Section 2.3.4, as well as two variables to identify the ID of the next available and next used descriptors. Each virtqueue can have multiple descriptors, so there is also the size of the virtqueue, which is given by the number of descriptors. Since each device can have more than one virtqueue it is necessary to add variables to be possible to identify where the base address of the virtqueue, the base address of the data section are, as well as the ID of the virtqueue.

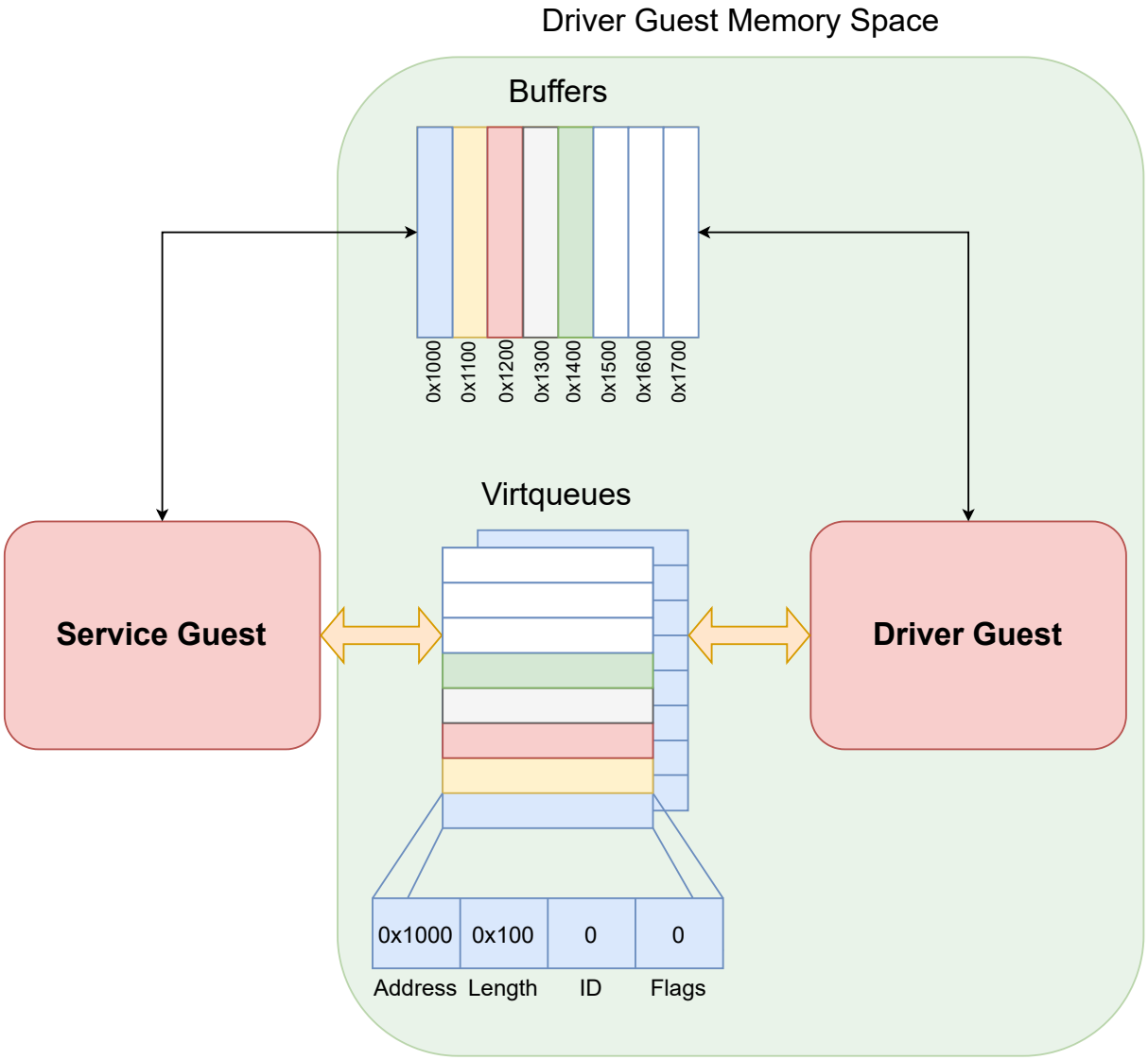


Figure 3.3: Interaction between guests and data structures.

As presented in Figure 3.3 each virtqueue contains an array of descriptors describing the buffers:

```
struct pvirtq_desc {
    uint64_t addr;
    uint32_t len;
    uint16_t id;
    uint16_t flags;
};
```

Listing 3.3: Descriptor structure.

3.1.2 Tests

This initial implementation is the starting point when it comes to communication between two isolated VMs. It is essential to ensure its correct functioning in order to move to the VirtIO implementation itself. With this in mind, the following tests were performed using the QEMU. QEMU is an open-source software that emulates one or more processors, allowing the virtualization of a system. This hypervisor is prominent in the embedded world due to its ability to handle several architectures. An advantage of the QEMU's usage is its support of the aarch64 architecture, which is the CPU architecture in which this dissertation work is implemented.

The first test consisted in guaranteeing that a message can travel from the driver guest to the service guest. So it was created a simple message in the driver guest. The front-end driver stores it in the memory and uses the implemented mechanism to transfer the data to the service guest. To guarantee the correct functioning, the service guest prints the descriptor as well as data that it describes. It is crucial to print all the descriptors values because a single corrupted parameter of a descriptor can lead to malfunctions even if the data is correct. This happened in multiple tests because the size of the descriptor was wrong since the memory addresses that are in the positions right after where the message is stored are not always empty. This means that if the size is wrong, it can print rubbish after the message, which is a massive problem when the goal is to send requests to a device. In some messages, it would not be possible to notice the bug, but by printing the descriptor size it would be obvious. This allowed to correct a bug in the mechanism.

To test the second implementation of this mechanism (the capacity to send data not only from the driver guest to the service guest but also in the opposite direction) a new test was necessary. Its first part is similar to the other test, but it is also crucial to test the transmission of data from the service guest to the driver guest. Thus, it was executed the test explained before and on top of that every time the service guest receives a new interrupt originated by a hypercall and it processes a descriptor, the service guest puts a new message ("Successful operation") in a different memory address. Afterward, the service guest creates a descriptor containing this message's information and stores the new descriptor in the memory. Thereafter, the back-end notifies the driver guest by calling a hypercall. The driver guest receives the interrupt that was generated by the hypercall, reads the new descriptor and reads the data. In the end, the driver guest prints the message as well as the descriptor so that it is possible to check if the message was received without errors and if the descriptor presents the correct values.

3.2 VirtIO

After establishing that it is possible to communicate between two virtual machines by using an interface that is similar to VirtIO it is initialized the VirtIO's interface implementation. As in the initial implementation with shared memory and hypercalls, this implementation uses at least a service guest. It is possible that there is more than one dedicated guest, this means that there can be more than one service guest where

each one contains the back-end driver for different devices. This implementation also allows a VM to have front-end and back-end drivers at the same time. To simplify, the design and implementation that will be presented only uses two VMs: the driver guest (front-end) and the service guest (back-end).

The following figures (Figure 3.4 and Figure 3.5) present the steps that are necessary to exchange data between the two guests. The former, explains the view of the front-end and the latter presents the view of the back-end.

With VirtIO it is possible to both transmit and receive data from a device by using different virtqueues. The mechanism is similar for both virtqueues. However, when the goal is to send data from the driver guest to the device, the driver guest creates a descriptor containing the information of the data (the address where it is stored, length and flags). When the goal is to receive data from the device, the driver guest creates a descriptor, which will describe a buffer with the maximum size possible for the device, since the driver does not have the knowledge of how big the data is. Thence, the service guest is responsible for reading the descriptor that was created by the driver guest and changing its values to describe the data that the device sends to the driver guest.

The action starts in the driver guest by creating a zeroed descriptor (the only variable that is set is the ID of the descriptor) and then storing the data in the memory. If the virtqueue's goal is to receive data, this last step is not taken. After that, the descriptor created is updated and then the flags are set. It is necessary to make the descriptor available, so it can be read in the back-end driver. Then, the descriptor is copied to the memory and some variables that are crucial to this mechanism are updated. The variable *next_avail* is used to store the ID of the last descriptor that was made available by the driver guest and the *driver_wrap_counter* is used to set the available and used flags of the descriptors. In the end, it is sent a notification to the back-end.

The back-end will process the descriptor and if everything goes as expected, it sends an interrupt back to the front-end. The device interrupt can be one of the following: Used Buffer Notification or Configuration Change Notification. The first one, as the name indicates, is when the device has used a buffer in at least a virtqueue. The second is used every time the configuration parameters of the physical device changes. To know the goal of the interrupt it is necessary to read the register *InterruptStatus*. If bit zero of this register is set, it means that it is a Used Buffer Notification. If bit one of the register is set, it is a Configuration Change Notification.

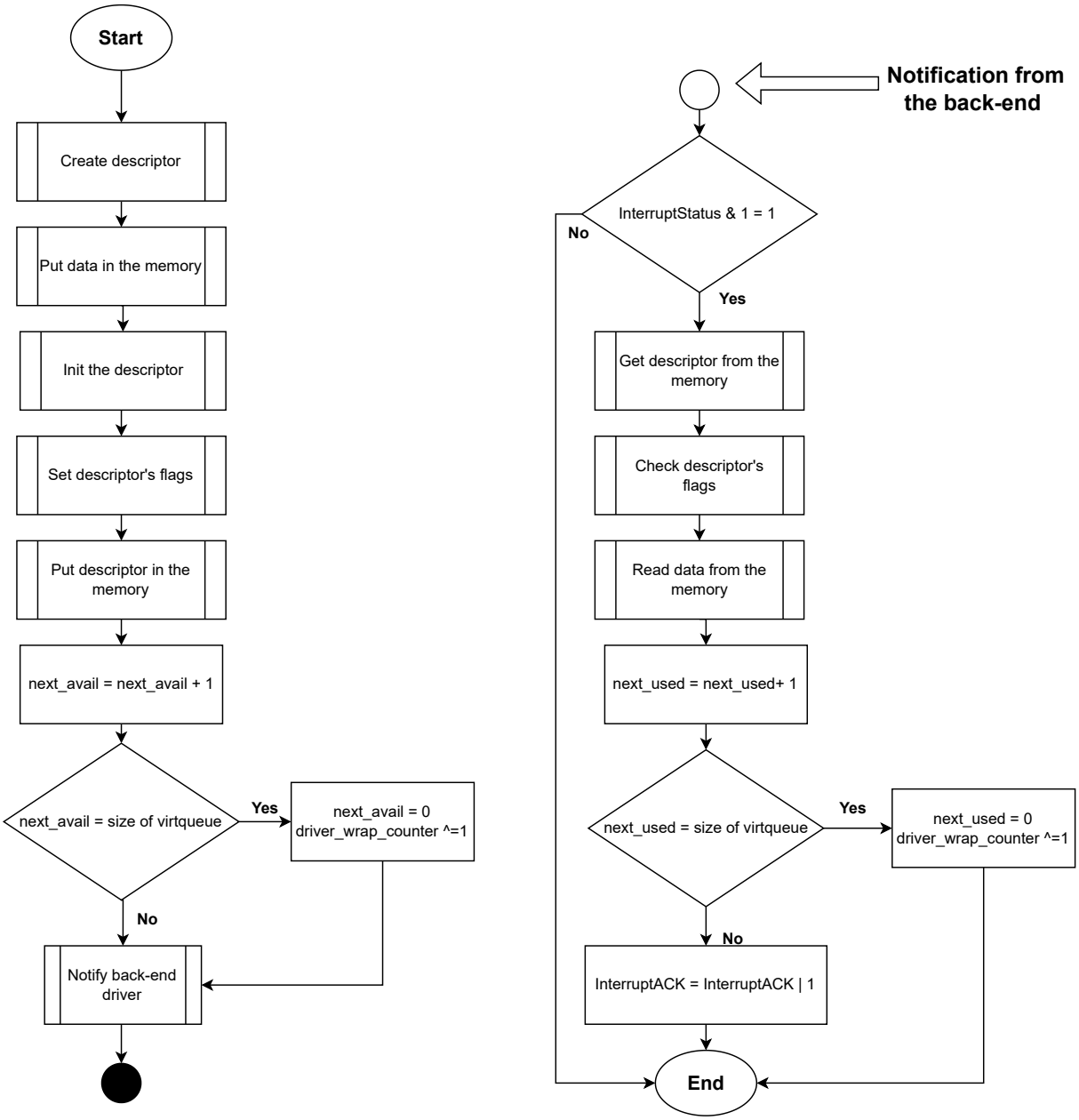


Figure 3.4: VirtIO main mechanism - front-end view.

In this case, the driver guest is waiting for a Used Buffer Notification. So, it is necessary to check if the bit zero is set and then the driver guest gets the descriptor from the memory. Afterward, it reads the flags of the descriptor to verify if the descriptor is used. If the goal is to receive data, the driver reads it from the memory using the descriptor information. Then, the driver guest updates internal variables and sets the bit of the event in the *InterruptACK* register, to notify the back-end, that the event that caused the interrupt has been handled.

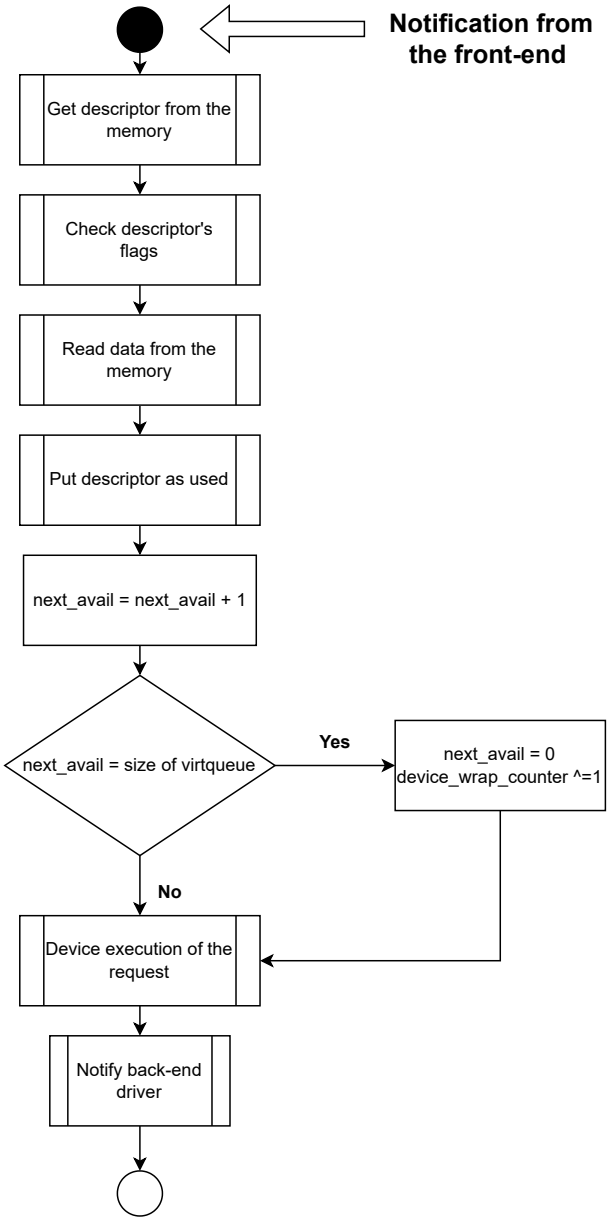


Figure 3.5: VirtIO main mechanism - back-end view.

As for the back-end view (Figure 3.5), when the front-end notifies the back-end, the first step is to get the descriptor. Then, the service guest needs to check if the descriptor is actually available and if this turns out to be true, it reads the data that the descriptor describes and marks the descriptor as used by changing its flags. In the end, the service guest executes the request and finally, it notifies the front-end driver. Note that the back-end driver only notifies the front-end when the device finishes its execution.

In the case of receive virtqueues, instead of reading data from the memory, the service guest stores the data sent by the device in the memory, then updates the values of the descriptor with the information of the data, puts it as used and notifies the front-end driver.

3.2.1 VirtIO Transport - MMIO

The hypervisor under which this dissertation is going to be implemented only supports MMIO, therefore, the transport will be implemented over MMIO. VirtIO devices that use MMIO provide a set of memory mapped registers (previouly presented in Section 2.3.5). These registers are all 32 bits wide. The device-specific configuration space can be accessed after the address offset 0x100. Here, it is possible to read or write to the registers using 8 bit wide accesses for 8 bit wide fields, 16 bit wide accesses for 16 bit wide fields and 32 bit wide accesses for 32 and 64 bit wide fields.

Each virtual machine that contains VirtIO devices can interact with their registers by accessing the memory addresses after the virtual address (base address) defined in the configuration. When one of these registers is accessed, it starts a mechanism of trap and emulation. There is a trap to the hypervisor, which will stop the virtual machine and emulate the device and then, the hypervisor is in charge of updating the value of the register according to the result of the operation. Section 3.2.5 explains this procedure in depth.

3.2.2 Virtqueues

As explained in Section 2.3.4 there are two types of virtqueue layouts: Split and Packed. However, as mentioned before in Section 2.3.4, the packed virtqueue layout was chosen due to the problems that the split virtqueue has.

In the packed virtqueue both front-end and back-end drivers have a single-bit wrap counter initialized to 1. The wrap counter maintained by the front-end driver is called Driver Wrap Counter and the wrap counter maintained by the back-end driver is named Device Wrap Counter. The Driver Wrap Counter changes its value every time the front-end driver makes the last descriptor available. The Device Wrap Counter changes its value each time the back-end driver uses the last descriptor. The Driver Ring Wrap Counter is equal to the Device Ring Wrap Counter when both front-end and back-end drivers are processing the same descriptor, or when all the available descriptors have already been used.

To mark a descriptor as available or used the following flags may be used:

```
1 #define VIRTQ_DESC_F_AVAIL (1 << 7)
2 #define VIRTQ_DESC_F_USED (1 << 15)
```

To mark a descriptor as available, the front-end driver needs to set the bit *VIRTQ_DESC_F_AVAIL* in the descriptor's flags to match the Driver Ring Wrap Counter and set the bit *VIRTQ_DESC_F_USED* bit to match the inverse value. This means that initially when the counter is 1, to make the descriptor available the front-end driver needs to set the bit *VIRTQ_DESC_F_AVAIL* to 1 and the bit *VIRTQ_DESC_F_USED* to 0. When the Driver Ring Wrap Counter is 0, the procedure is the inverse, the bit *VIRTQ_DESC_F_AVAIL* is set to 0 and the bit *VIRTQ_DESC_F_USED* to 1.

To mark a descriptor as used, the back-end driver needs to set the bit *VIRTQ_DESC_F_USED* in the descriptor's flags to match the internal Device Ring Wrap Counter and the bit *VIRTQ_DESC_F_AVAIL* gets

the same value. This means that initially when the counter is 1, to make the descriptor used, the back-end driver needs to set both bits *VIRTQ_DESC_F_AVAIL* and *VIRTQ_DESC_F_USED* to 1 and when the Device Ring Wrap Counter is 0, the bits *VIRTQ_DESC_F_AVAIL* and *VIRTQ_DESC_F_USED* are set to 0.

Therefore, if the bits *VIRTQ_DESC_F_AVAIL* and *VIRTQ_DESC_F_USED* are different, it means that the descriptor is available. If they are equal, it means that the descriptor is used.

To configure the virtqueues, the driver needs to carry out the following steps during initialization:

1. Initially, it is necessary to select the virtqueue by writing its index into the register *QueueSel*;
2. Then, it is essential to check if the virtqueue is not in use. To do that, the register *QueueReady* is read and the value zero is expected;
3. After that, it is read the register *QueueNumMax* to get the maximum number of elements that the queue can support. If the value is zero, it means that the queue is not available;
4. Now that it is known that the virtqueue is available it is crucial to allocate memory for the virtqueue and zero it;
5. Afterward, the front-end writes the size of the virtqueue in the register *QueueNum*, so that the back-end driver can know the size of the queue;
6. It is necessary to write the guest physical addresses of the virtqueue, the driver and device areas into the pairs of registers *QueueDescLow/ QueueDescHigh*, *QueueDriverLow/ QueueDriverHigh* and *QueueDeviceLow/ QueueDeviceHigh*;
7. The final step is to put the register *QueueReady* to 1, to notify the back-end that it can execute the request from this virtqueue.

3.2.3 Feature Bits Negotiation

The feature bits are essential to assuring compatibility between different versions of devices and drivers. These features are negotiated using MMIO registers, where each bit represents a feature. For instance, if the back-end driver has a new feature, the bit correspondent to this feature is set. If the front-end driver is not recent and does not support this new feature, it will not write that bit back to the device because it does not support the feature. The same happens in the opposite direction. If the front-end driver has a new feature that the back-end does not support, the back-end will not offer this feature.

The feature bits are divided into three sections:

- 0 to 23: Feature bits for the specific device type. This means that the same bit has different meanings according to the device type. For instance, the first bit in a console device is the bit *VIRTIO_CONSOLE_F_SIZE* which means that a specific configuration (columns and rows) is valid, while in a network device the same bit is *VIRTIO_NET_F_CSUM* which means that the device

supports packets with a partial checksum. These bits are usually used to indicate if some fields of the device configuration space are going to be used;

- 24 to 37: Feature bits reserved for extensions to the queue and feature negotiation mechanisms. These feature bits are device-independent feature bits, which means that they are equal for all types of devices;

- 38 and above: Feature bits reserved for future extensions [24].

In this dissertation's implementation, there are two mandatory feature bits. The *VIRTIO_F_RING_PACKED* and *VIRTIO_F_VERSION_1*. The first indicates that there is support for the packed virtqueue layout and the second identifies if there is compliance with version one of the VirtIO's specification.

Every back-end of a VirtIO device offers the features that it supports to the front-end. During the device initialization (Section 3.2.4), the front-end reads these features, compares with the ones that it is compatible and sends back to the back-end the subset that it accepts. The only way to renegotiate is to reset the device, so if there is a problem in this negotiation, it is not possible to use the device.

The front-end driver must not accept any feature that the device did not offer. Moreover, if the back-end driver does not offer a feature that the front-end understands, then the front-end driver should go into backwards compatibility mode. If not, it must stop the initialization and set the *FAILED (128)* device status bit.

In what concerns the back-end driver, it should accept any subset of features that the driver accepts. If this does not happen, it will fail to set the *FEATURES_OK* device status bit when the driver writes it and when the driver re-reads it to confirm its value the bit is going to be zero and the driver concludes that the negotiation failed.

Figure 3.6 presents the sequence of events that the front-end driver must follow to negotiate the feature bits. To negotiate, the features needed are four 32-bit MMIO registers:

- *DeviceFeatures* - Flags that represent the feature bits that the back-end driver supports. By reading this register, the front-end driver gets 32 consecutive features bits supported by the back-end. If the last value written to the register *DeviceFeaturesSel* is 0, it gets the bits 0 to 31. If the register *DeviceFeaturesSel* is 1, then it gets the feature bits 32 to 63;

- *DeviceFeaturesSel* - Register used to select which set of 32 bits of the feature bits will be returned when reading the register *DeviceFeatures*;

- *DriverFeatures* - Flags that represent the feature bits that the front-end driver supports and accepts. This register follows a mechanism similar to the register *DeviceFeatures*. If the last value written to the register *DriverFeaturesSel* is 0, it gets the bits 0 to 31. If the register *DriverFeaturesSel* is 1, then it gets the feature bits 32 to 63;

- *DriverFeaturesSel* - Register used to select which set of 32 bits of feature bits will be accessible by writing to the register *DriverFeatures*.

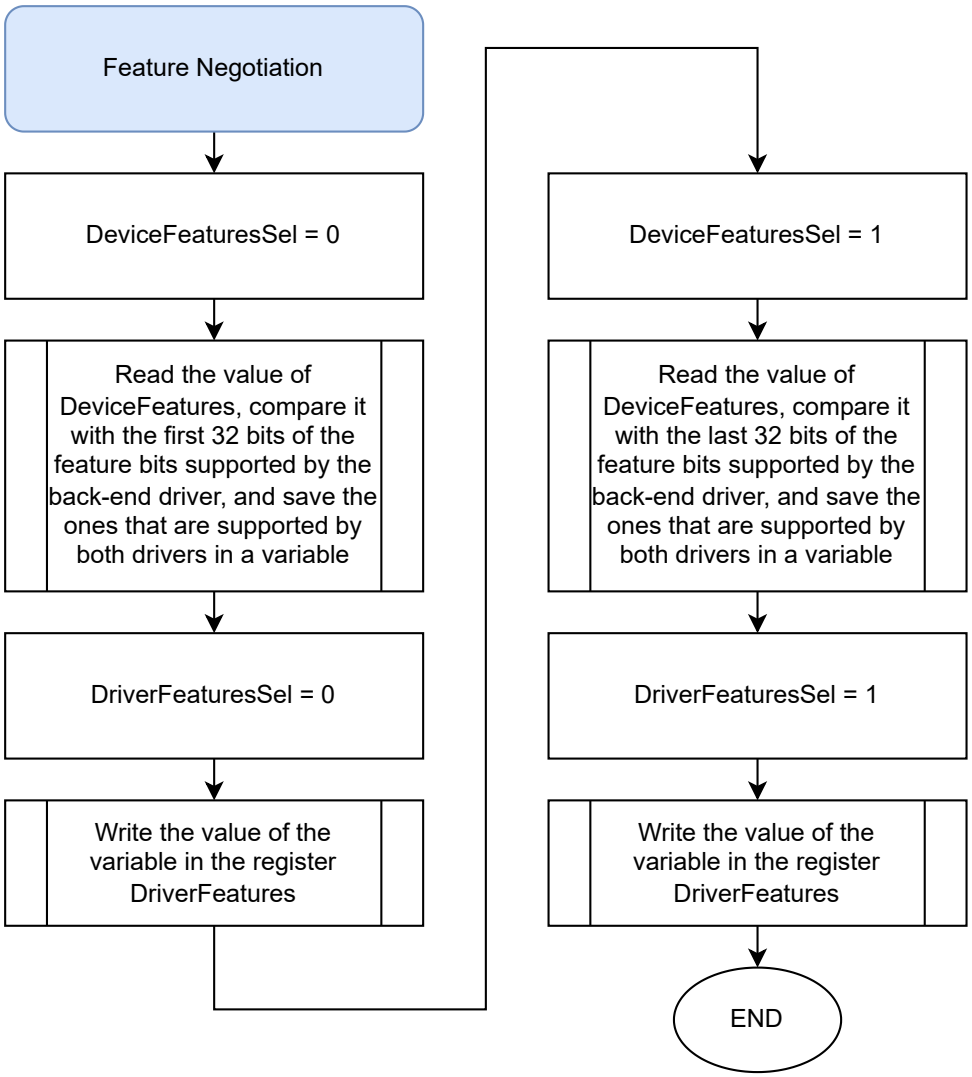


Figure 3.6: Feature Bits Negotiation executed by the front-end driver.

The first step of the negotiation is to put the register *DeviceFeaturesSel* as 0 and read the value of the register *DeviceFeatures* that will return the 32 least significant bits of the feature bits the back-end driver supports. Then, the front-end driver compares these bits with the 32 least significant bits of the feature bits that the front-end supports and stores the intersection in a variable that will be written into the register *DriverFeatures* right after putting the register *DriverFeaturesSel* as 0. After that, the same process is executed, for the most significant 32 bits of the feature bits. For this, it is only necessary to put the registers *DeviceFeaturesSel* and *DriverFeaturesSel* as 1 instead of 0.

3.2.4 Device Initialization

The initial stage of the initialization of a device is the same, whatever type it is. The specification defines a set of operations that are crucial to make sure the driver guest initializes the device as it is supposed to prevent errors. This stage is also important for compatibility issues, since it is here where the features that will be used are defined. Figure 3.7 presents the steps that are necessary for the initialization.

Note the variable *Status*. This variable is the device status, which provides an indication of the device initialization's steps that were already concluded. To get the steps, the bits of the register should be read. If the bit it is set means that the step was completed.

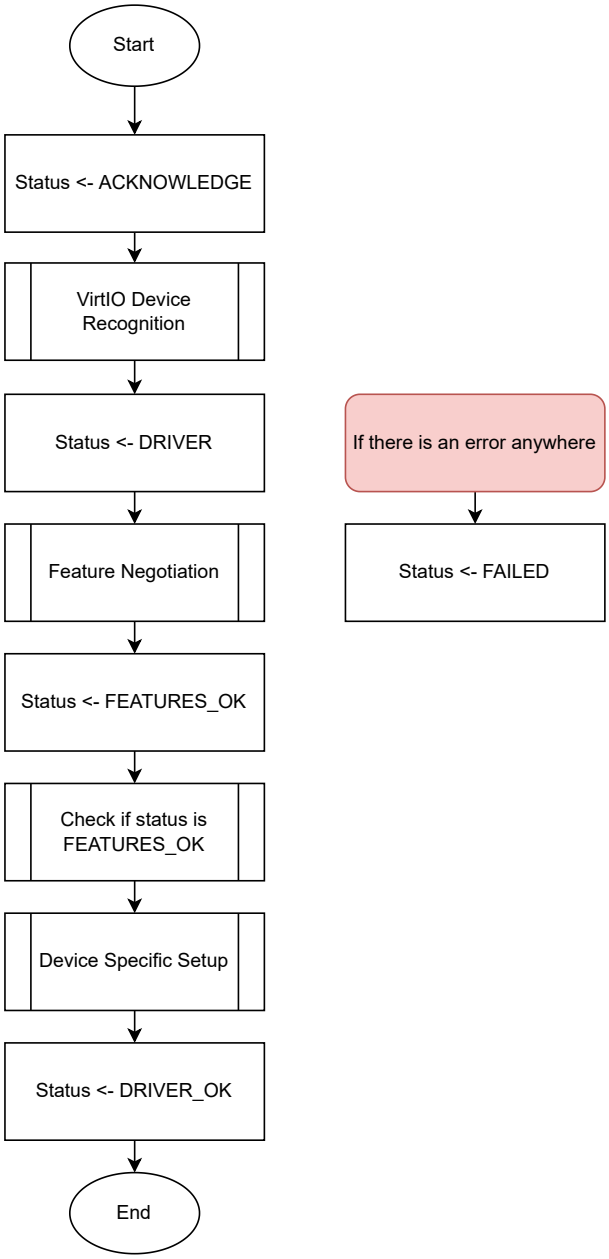


Figure 3.7: Device Initialization.

The bits that can be set are the following:

- If the first bit is set, *ACKNOWLEDGE (1)*, it means that the guest found the device and it recognized the device as a valid VirtIO device;
- The second bit, *DRIVER (2)*, indicates that the guest has a driver compatible with the VirtIO device;
- The bit *DRIVER_OK (4)* specifies that the driver is ready to drive the VirtIO device;
- The bit *FEATURES_OK (8)* indicates that the driver has accepted all the features it understands and that the feature negotiation is complete;
- The bit *DEVICE_NEEDS_RESET (64)* means that there was an error from which the device cannot recover and it is necessary to reset the device;
- The bit *FAILED (128)* reveals that the guest has given up on the device since something went wrong with it. It can be an internal error, a fatal error during the device operation, etc.

To initialize the device, the front-end driver must reset the device and right after that set the bit *ACKNOWLEDGE (1)* in the device status field to indicate that the OS has already noticed the device and recognized it as a compatible device. Thereafter, the driver must set the bit *DRIVER (2)*, meaning that it can drive the device. After setting these two bits, it starts the feature negotiation, which is explained in detail in Section 2.3.2. After this negotiation, the bit *FEATURES_OK (8)* is set if all the features were understood and the feature negotiation was completed. Then, the device status is re-read to make sure that the *FEATURES_OK (8)* bit is still set. Otherwise, it means that the device does not support the driver's subset of features, meaning that the device is unusable. Afterward, the driver performs a device-specific setup, including the discovery of the virtqueues that will be used, reading and writing the configuration space of the VirtIO device and populating the virtqueues. Finally, it is necessary to set the *DRIVER_OK (4)* bit to indicate that the driver is set up and ready to be used. If any of the previous steps go irrecoverably wrong, the driver must set the bit *FAILED (128)* and it can reset the device later if desired. In this case, the initialization will stop and the device cannot be used [24].

3.2.5 Implementation

This VirtIO implementation does not use shared memory to store the descriptors and the buffers. The driver guest allocates a memory region and the service guest access it. However, Bao's original implementation does not allow a VM to access to the memory of another VM. With this in mind, it was necessary to change Bao's code to allow a guest to access another guest's memory region, meaning that the memory isolation can disappear. This way, the service guest can access all the descriptors as well as the buffers that are place in the driver guest's memory.

The implementation was done to be possible to have the functions of the back-end driver and front-end driver in the same files. This way this files can be used as a VirtIO library. For this reason, some of the function that will be presented have the same name, but with the words “back-end” or “front-end” in it because they have the same purpose. However, they manipulate different variables according to the driver where they are being used.

Some structs are similar to the ones presented in Section 3.1.1. For instance, the struct *pvirtq_desc* is exactly the same. However, there was the need to change the other structs, such as the *virtio_device*:

```
struct virtio_device {
    volatile struct virtio_mmio_reg *mmio_reg;
    int vq_number;
    deviceType_e type;
    int irq_flag;
    int device_flag;
    long id;
    uint64_t feature_bits;
    uint64_t negotiated_feature_bits;
    char data[50];
    char received_data[50];
};
```

Listing 3.4: VirtIO device structure.

In this struct, it was created a new variable (*mmio_reg*) to store the value of the MMIO registers in the back-end driver or to access them in the front-end driver.

This implementation uses Bao’s IPC mechanism to notify the service guest when there is a new descriptor. For that reason the variable *irq_flag* that is incremented every time the back-end receives an interrupt from Bao was created. The back-end can also receive interrupts from the physical device, which is why it was created a variable that is incremented every time the back-end driver receives an interrupt from the device (*device_flag*).

Feature negotiation is essential to ensure the compatibility between different devices and VirtIO versions and implementations. There are two variables to be possible to negotiate the features bits. The *feature_bits* that contains the feature bits that the driver is compatible with and the *negotiated_feature_bits* that will get the feature bits that were negotiated between the front-end and the back-end. Finally, the back-end can receive data from the physical device, so it was necessary to create an array to store this data (*received_data*).

The struct of the virtqueue is almost the same as the one presented in Section 3.1.1, but the two variables that refer to the memory addresses (*shmem_queue_base* and *shmem_data_base*) were replaced by a new struct (*vq_mem*). This new struct also contains variables that define the size of the descriptor and data sections, as well as the last address where the data of the current virtqueue was stored:

```
struct virtq {
    long used_wrap_count;
```

```
3     long next_used;
4     long avail_wrap_count;
5     long next_avail;
6     long size;
7     struct vq_mem mem;
8     long queue_num_max;
9     char queue_ready;
10    short id;
11    /* Descriptors */
12    struct pvirtq_desc *desc;
13 };
```

Listing 3.5: Virtqueue structure.

Moreover, it was necessary to add two new variables (queue_num_max and queue_ready) to store the values that can be changed by the front-end driver, using the MMIO registers.

In the VirtIO implementation, the following functions were created:

- *virtio_device_init* - This function is the first to be called and it is responsible for the initialization of the all variables of the device (struct *virtio_device*);
- *virtio_front_end_mmio_init_before_config* - This function is called in the front-end driver right after the function *virtio_device_init*. It is in this function where the values of the registers *MagicValue*, *Version* and *DeviceID* are checked. Furthermore, it is here where the function responsible for the feature bits negotiation is called;
- *virtio_front_end_mmio_init_after_config* - This function is called after the definition of the config space parameters. It is responsible for initializing the virtqueues, one by one, communicating with the back-end using the MMIO registers associated with the virtqueues;
- *virtio_back_end_mmio_init* - The function that corresponds to the last two functions presented for the back-end driver has the same purpose, initialize the device. However, this function only has to clear the MMIO registers, set the *MagicValue*, the *Version* and the *DeviceID* to the appropriate values and then initialize the virtqueues;
- *virtio_front_end_queue_init* and *virtio_back_end_queue_init* are the functions for the front-end and back-end, respectively, which are responsible for the initialization of a virtqueue. These functions are called every time it is necessary to initialize a new virtqueue. They both call a more generic function (*virtio_queue_init*). However, in the back-end it is essential to clear the memory addresses of the descriptors and data sections;
- *virtio_front_end_feature_bits_negotiation* and *virtio_back_end_feature_bits _negotiation* are used for the feature bit negotiation. In the front-end driver, the function to negotiate the features is called

at the initialization. On the other hand, in the back-end, it is called when the register *DriverFeatures* is accessed by the driver guest;

- *virtio_front_end_mem_data_write* and *virtio_back_end_mem_data_write* are the general functions used to put a descriptor in the memory. The front-end is responsible for the creation of new descriptors, so the front-end function is also responsible for the creation of a descriptor before storing it in the memory. This is done using the function *virtio_desc_init*. The back-end driver will only overwrite the descriptor that was created by the front-end and that is why it does not have to create any descriptor;

- *virtio_front_end_queue_desc_write* and *virtio_back_end_queue_desc_write* are the functions that set the flags of a descriptor and put it in the memory by calling the functions (*virtio_front_end_mem_desc_write* and *virtio_back_end_mem_desc_write*). These functions are similar except for the wrap count that is used (avail wrap count in the front-end and used wrap count in the back-end), as well as the virtqueue counter (*next_avail* in the front-end and *next_used* in the back-end).

  In the end of the function it is necessary to notify the other guest that the process is concluded. The front-end only needs to write the ID of the virtqueue that is being used in the *QueueNotify* MMIO register and the back-end uses a VirtIO hypercall (will be explained later in Section 3.3.1) to generate an interrupt to the driver guest:

```
1    #define VIRTIO_NOTIFY_FRONT_END(dev_id) virtio_hypercall(dev_id,
     0, INTERRUPT_OP, 0)
2    #define VIRTIO_NOTIFY_BACK_END(vq_id) device->mmio_reg->
     QueueNotify = vq_id
3
```

- *virtio_front_end_queue_desc_read* and *virtio_back_end_queue_desc_read* are equivalent to the last two functions, but their goal is to read the descriptor instead of writing it. Both functions need to verify the flags to guarantee that the descriptor was not corrupted or contains any error and then, both of them read the data of the memory address that was received in the descriptor;

- *virtio_front_end_mem_data_read* and *virtio_back_end_mem_data_read* are in charge of copying the data of the memory address that is in the descriptor to an internal space. After these functions, the back-end can send this data to the physical device and the process is concluded.

3.2.6 Tests

After implementing the VirtIO interface, it was tested if the virtqueues were working as expected and if the data was being transferred between two guest without suffering any problem in its integrity.

Therefore, two groups of tests were executed. The first group's goal was to test the communication. Initially, it was executed a test similar to the second test presented in Section 3.1.2. However, in this test it is used not only a virtqueue but two virtqueues: the transmit virtqueue (to transfer data from the driver guest to the service guest) and the receive virtqueue (to transfer data from the service guest to the driver guest). The mechanism starts with the driver guest creating a message and a descriptor with its information. Then, it puts the descriptor in the transmit virtqueue and notifies the service guest by writing to the register *QueueNotify*. The service guest receives an interrupt and processes the descriptor. Afterward, the service guest prints the message and descriptor to be possible to verify their content. After that, it creates a new message and a new descriptor to test the other virtqueue. It puts the new descriptor in the receive virtqueue and notifies the other guest. Finally, the driver guest prints the descriptor that is in the receive queue as well as the message sent by the service guest.

It was essential to execute this test with multiple transmissions to make sure that the mechanism was capable of queuing more than one descriptor in the virtqueue. An important issue in this test is the value of the flags. It is essential to guarantee that their values are correct, especially the two flags that define if the descriptor is available or used (*VIRTQ_DESC_F_AVAI* and *VIRTQ_DESC_F_USED* respectively).

The second group of tests was conducted after the complete success of the previous group of tests. Their goal was to simulate a device behaviour by creating an internal array in the service guest which receives requests from the driver guest. The driver guest can send messages to write or read a position of the array. When the service guest receives an interrupt, it reads the descriptor from the transmit virtqueue and processes the request. Thenceforth, it updates the array according to the operation and then sends the value if the operation is to read or if the operation is to write a value, it sends a message declaring that the operation was well succeeded. Finally, the service guest prints all the values of the array, to show that the values were updated and the driver guest prints the message received from the service guest.

These two groups of tests were initially executed using QEMU to simulate the environment and then using a Xilinx board, since the final goal is to run the VirtIO interface in this board. It was used the Zynq UltraScale+ MPSoC. It combines feature-rich 64-bit quad-core processing system (PS) and a Xilinx programmable logic (PL) UltraScale architecture in a single chip. Besides, it also includes an on-chip memory, multiport external memory interfaces as well as a large set of peripheral connectivity interfaces such as UART or ethernet (GEM).

When testing in QEMU, all the tests were well succeeded. However, when it was migrated to the board, the system did not work. The issue was the *cpu_idle* that was implemented to power down the CPU. When emulating in QEMU the CPUs are not physical and can not be powered down. When using the board, the CPUs are powered down and their registers are lost. This means that when Bao puts a CPU in idle state, it does not know the last executed instruction and the system fails. To solve this problem it was necessary

to change Bao's code and instead of calling the function to power down the CPU it puts the CPU in the wfi, i.e., waiting for interrupts, so that, the CPU never stops totally, only waits for a new interrupt, meaning that the registers are intact and at the same time it does not execute any instruction until Bao sends an interrupt.

3.3 VirtIO Integration on Bao Hypervisor

The final step is to implement VirtIO on Bao, because even though the VirtIO interface that deals with the data is not implemented on the hypervisor, which is uncommon in many popular hypervisors, it is needful to create compatibility. One of the aspects is the transport of the VirtIO. This hypervisor uses MMIO as transport, so the VirtIO was implemented using MMIO.

Figure 3.8 exhibits VirtIO's architecture. Both front-end and back-end drivers are located in VMs, not in the hypervisor. Note that when using VirtIO, the service guest is the only VM that can directly access physical devices. The mechanism allows the existence of back-end drivers and front-end drivers in the same VM. However, the simplest scenario is when there is one service guest with all the back-end drivers of all the devices that will be used.

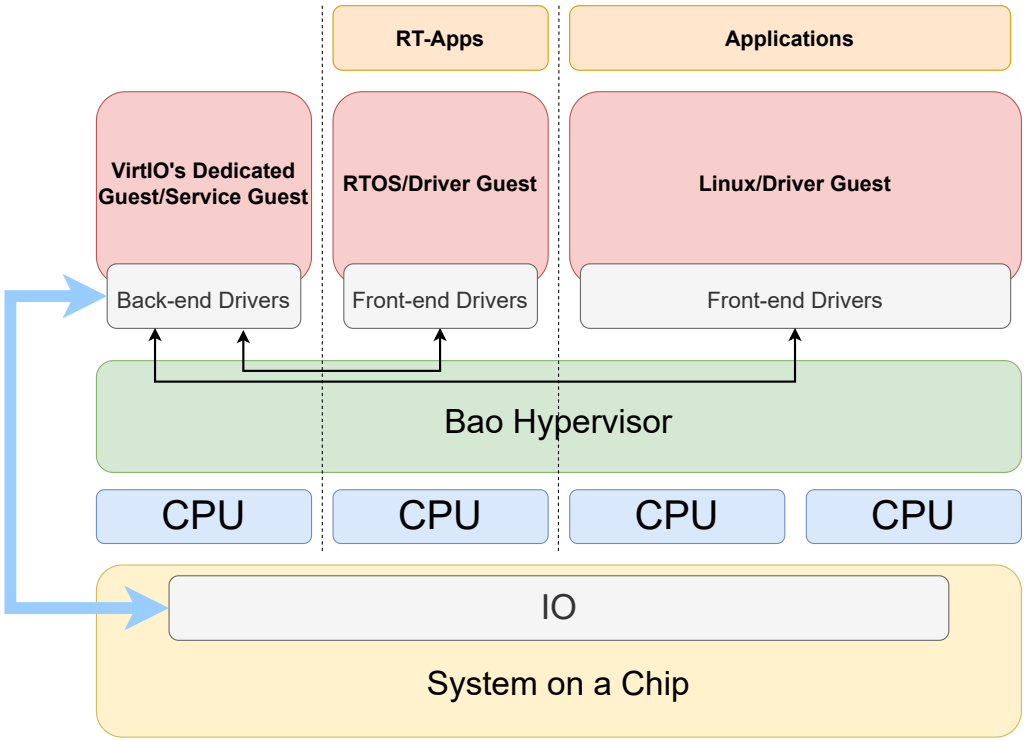


Figure 3.8: VirtIO implementation on Bao.

When the front-end driver of a VirtIO device wants to access its MMIO registers, it reads or writes in the correspondent virtual address. This generates a trap to the hypervisor that will process the action by emulating the device.

The back-end driver can have one of two operation modes: polling or interrupts. The first mode, polling, is constantly invoking VirtIO hypercalls to query the hypervisor if there was any new trap. The second mode, interrupts, waits for an interrupt that is generated by the hypervisor.

When the hypervisor processes the action that generated the trap, it notifies the service guest and it will read or write a register depending on the action. At the same time, the hypervisor puts the CPU that runs the driver guest in the idle state, waiting for a response from the service guest. Then, the service guest will use, once more, the VirtIO hypercall, this time to send the value of the register. After that, in the case of a read, the hypervisor will put the value read, in the register of the front-end. Finally, the driver guest can continue its execution until there is a new try to access to a MMIO register that belongs to a VirtIO device.

Figure 3.9 presents the big picture of the mechanism used to implement VirtIO on Bao.

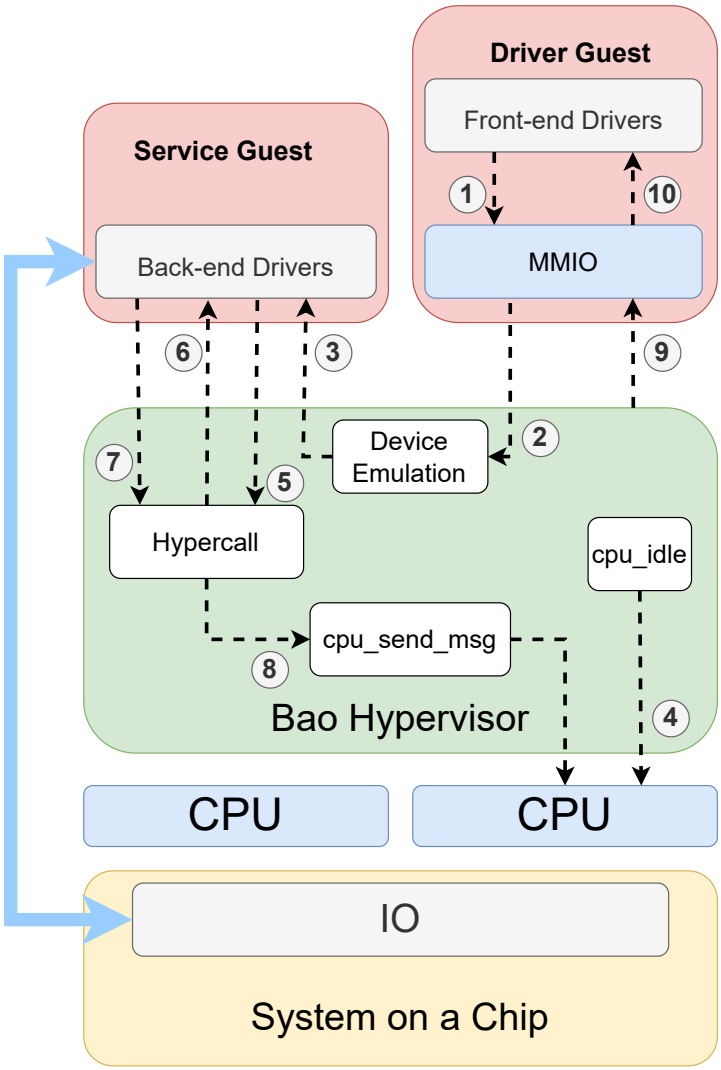


Figure 3.9: VirtIO mechanism on Bao.

1. The front-end driver of the Driver Guest tries to access an MMIO register that belongs to a VirtIO device;

2. After that, there is a trap to Bao that decodes the access and sends the information to the back-end driver that emulates the device. Here, the hypervisor processes the information of the access and updates the variables;

3. Then, the hypervisor notifies the service guest that there was an access to a VirtIO device's MMIO register. If the operation mode of the device is not by interrupts, the mechanism is slightly different. Instead of a notification, the ID of the device is stored in a list of IDs and later the service guest asks if there was any access;

4. The CPU should be put in the idle state until there is a response from the service guest. Although it causes the system to be slower, it is essential that the guest waits while the device is being emulated, otherwise, the system could fail;

5. Then, the service guest uses the VirtIO hypercall to request information regarding the guest access, such as the access address, if the access was a read or write, width of the access, etc.;

6. The hypervisor receives the hypercall and returns all the information necessary to the service guest, which will process that data and act accordingly;

7. Afterward, the service guest uses the hypercall again to send back to the hypervisor the value of the register that was accessed by the driver guest in case of a read or just a confirmation in case of a writing;

8. After that, it is necessary to "wake up" the other CPU (the one that was running the driver guest and it is currently in the idle state). For that, it is used the function cpu_send_msg;

9. The register that was accessed must now be updated so that the driver guest can continue its execution;

10. Finally, the front-end driver reads the value of the register and continues the execution as if nothing happened.

3.3.1 Implementation

This section goes through the steps that were taken to make Bao compatible with the VirtIO interface. It will be explained the mechanism presented before taking into account Bao's infrastructure.

Foremost, it was necessary to create parameters to enable configuring everything that is necessary to use VirtIO on Bao. These parameters are divided into two groups: one for the general config and one for the config of each virtual machine.

The first group is in charge of the description of the VirtIO devices that will be used in the entire system. This means that it has to define the number of devices as well as information of the shared memory if they use it.

If there is the need to use shared memory to store the virtqueues and the data that they describe, of a device, the following parameters must be set:

- shmem_id: to define what is the shared memory ID that is going to be used;
- shmem_base: that defines what is the physical address where the shared memory starts;
- shmem_size: to define the size of the shard memory.

The second group is in charge of the configurations necessary for each VM, such as:

- va: to define the virtual address that is going to be used to access the MMIO registers of the device;
- size: to determine the size of the MMIO region (usually 0x200);
- interrupt: to set the interrupt that is going to be generated. It is only required for the back-ends and only has significance when the parameter *polling* is not true;
- device_id: to choose which device the VM is going to use;
- is_back_end: to specify if the VM is going to contain the VirtIO back-end driver for the VirtIO device that corresponds to the *device_id*;
- polling: to delineate if the execution mode is going to be polling or by interrupts. This parameter is only required for the back-ends.

With this in mind, the following structure was created to group all the parameters necessary to configure the VirtIO:

```
1  struct virtiodevice{
2      uint64_t shmem_id;
3      uint64_t shmem_base;
4      size_t shmem_size;
5
6      uint64_t va;
7      size_t size;
8      irqid_t interrupt;
9      uint32_t device_id;
10     int frontend_id;
11     int backend_id;
12     bool is_back_end;
13     bool polling;
```

```
};
```

Listing 3.6: VirtIO device configuration structure.

Note the existence of two parameters that were not explained (*frontend_id* and *backend_id*). These two parameters can be defined in the config, although not recommended since it can be more prone to errors. Each device must have a back-end driver and a front-end driver. These two variables contain the ID of the virtual machine where the front-end driver and the back-driver are located. If the user defines these parameters, their values will be checked in the initialization and then, if the VMs that were chosen do have the front-end and back-end drivers of the same device, the initialization will continue. Otherwise, an error will be generated.

If these parameters are not configured, they will be generated automatically in the initialization by the function *virtio_linkage_init*. To do this, it is necessary to go through the config and search for all the VMs that have VirtIO devices. Then, in each VM it is checked the config parameter *is_back_end* for each device and if this variable is true then the parameter *backend_id* is updated to the ID of the VM. Though, if the variable *is_back_end* is false or not defined in the config, the parameter *frontend_id* gets the value of the VM's ID.

Below, there is an example of the first part of the configurations needed to use the VirtIO on Bao. This configuration uses two VirtIO devices that use shared memory to store both the virtqueues and the device data.

```
.virtiodevicelist_size = 2,
.virtiodevicelist = (struct virtiodevice[]) {
    [0] = {
        .shmem_id = 0,
        .shmem_base = 0x70000000,
        .shmem_size = 0x00020000
        },
    [1] = {
        .shmem_id = 1,
        .shmem_base = 0x70030000,
        .shmem_size = 0x00020000
        }
}
```

Listing 3.7: Bao VirtIO device configuration.

Inside the configuration of the service guest, it is crucial to configure the two devices. Since this is the VM where the back-end drivers of both devices are placed, it is necessary to put the parameter *is_back_end* as true. As it is noticeable in the excerpt of code below, the device zero operates with interrupts, using the interrupt fifty-two, whilst the device one operates by polling and because of that the parameter *polling* is true in this device.

```
.virtiodevices_num = 2,
```

```
.virtiodevices = (struct virtiodevice[]) {
    {
        .device_id = 0,
        .is_back_end = true,
        .interrupt = 52,
    },
    {
        .device_id = 1,
        .is_back_end = true,
        .polling = true
    }
}
```

Listing 3.8: VirtIO device configuration for the service guest.

As for the driver guest, it is only needed to configure the size of the MMIO region of the device, choose the device ID and, finally, define the virtual address that will be used to access the MMIO registers:

```
.virtiodevices_num = 2,
.virtiodevices = (struct virtiodevice[]) {
    {
        .size = 0x200,
        .device_id = 0,
        .va = 0x1200000
    },
    {
        .size = 0x200,
        .device_id = 1,
        .va = 0x1202000
    }
}
```

Listing 3.9: VirtIO device configuration for the driver guest.

Configuration explained, it is time to pass to the initialization. In the initialization, Bao runs several functions that allow the proper initialization of the VirtIO's mechanism. The first, *virtio_linkage_init* is in charge of linking the VirtIO back-end drivers with front-end drivers as well as creating a list of devices (*virtiodevicelist*) that is crucial to the mechanism. Each element of this list contains the following struct:

```
struct virtiodeviceparams{
    node_t node;
    uint64_t id;
    unsigned long reg_off;
    unsigned long access_width;
    unsigned long op;
    unsigned long value;
    unsigned int cpu_id;
```

```
9       unsigned int sg_cpu_id;
10      unsigned long reg;
11  };
```

Listing 3.10: Structure with the parameters of a VirtIO device in Bao.

This struct incorporates all the parameters that will be used in the mechanism. The device ID, the *reg_off* that gives the offset of the MMIO register that was accessed, the *access_width* that indicates what was the width of the access to the MMIO register. This parameter is essential to detect errors because VirtIO MMIO only allow 4-byte wide and aligned accesses. Furthermore, the struct contains the operation, this is, if the access is a write or a read. The *value* is used to store the value that was written in the case of a write and to receive the value of the register that the service guest sends in case of a read. This struct also stores the ID of the CPU that accessed the MMIO register, as well as the service guest CPU ID (*sg_cpu_id*). This parameter is used to identify the CPU where the service guest is running in order to be possible send messages to it. Finally, the *ref* is used to store the CPU register that was used to store the value of the MMIO register.

It is in the function *virtio_linkage_init* where the *pollingdevicelist*, a list that contains the IDs of the devices that operate by polling that got a MMIO register accessed, is created. Meaning that every time a front-end of a device that does not use interrupts access to a MMIO register, Bao will not inject an interrupt to the service guest. Instead, it will store the device ID in this list and the service guest will ask if there was an access to the MMIO registers.

While the function *virtio_linkage_init* runs only in the CPU master, which means that is executed in just a CPU, every function of the initialization from now on is executed in all the CPUs.

Afterward, it is executed the function *vm_init* that runs the initialization of a VM. Here is called the function *vm_init_virtio* that verifies if the VM contains any VirtIO device. If that is true, it will copy some information from the config, such as the number of VirtIO devices that are in the VM and the devices' configs themselves, to a struct that stores information of the VM. Then, it will go through the devices of the VM and create an emulation object for each of them. This is what makes it possible to emulate the device every time a driver guest tries to access an MMIO register. Finally, it is called the function *virtio_serviceguest_cpu_init* that is responsible for updating the parameter *sg_cpu_id* of the *virtiodevicelist* with the value of the CPU's ID that is running the function. This is the end of the initialization of the VirtIO's mechanism on Bao.

Now that the mechanism is set to go, the virtual machines will start their execution and when a driver guest tries to access an MMIO register that belongs to a VirtIO device a trap is generated to Bao. That is when the execution goes from the VM to the hypervisor that will process the access and then it will run the function *virtio_mmio_emul_handler* which is the handler for every access to the MMIO register that belongs to VirtIO devices. This handler receives a struct containing information about the access, such as if it was writing or reading the value of the register, the width of the access, the register offset, etc.

At the beginning of the function, it walks through the list of VirtIO devices of the virtual machine defined in the configuration and checks if the address that was accessed actually belongs to a VirtIO device that

was correctly initialized. If it does not turn out to be true, it means that something went wrong and the function returns. If the device was found, it is necessary to copy the data of the access received in the function (CPU ID, register offset, CPU register that was used, operation and value) to the correspondent device in the *virtiodevicelist*. If the driver guest tried to write the register then it is necessary to read the value. If not, the parameter *value* is 0. In the end, is created a CPU message (*cpu_msg*) to inform the CPU that is running the service guest, that something happened. The message created needs an ID to identify the message as a VirtIO CPU message (*VIRTIO_CPUMSG_ID*). It is important to check the mode of operation of the device (interrupts or polling). This will change the type of event that is going to be generated (*VIRTIO_NOTIFY_SG* in the case of using interrupts and *VIRTIO_NOTIFY_SG_POLLING* in the case of not using interrupts). In the end, it is sent the CPU message and then the CPU enters in idle state. This is imperative, the CPU that runs the driver guest must enter an idle state and wait for a CPU message.

Now, in the CPU that runs the service guest, it is received a CPU message with the *VIRTIO_CPUMSG_ID*. This ID is connected to a function, *virtio_handler*, that will handle all the CPU messages associated with VirtIO. This function receives two parameters: the event that is defined in the creation of the CPU message and will be used to choose the function that will be called and the data that is the ID of the VirtIO device.

```
static void virtio_handler(uint32_t event, uint64_t data)
{
    switch(event){
        case VIRTIO_NOTIFY_SG:
            virtio_notify_serviceguest_handler(data);
        break;
    case VIRTIO_NOTIFY_SG_POLLING:
            virtio_insert_to_polling_queue(data);
        break;
        case VIRTIO_READ_NOTIFY:
        case VIRTIO_WRITE_NOTIFY:
            virtio_cpu_msg_handler(event, data);
        break;
    }
}
```

Listing 3.11: Bao function that handles CPU messages.

In this case, the event can be *VIRTIO_NOTIFY_SG* or *VIRTIO_NOTIFY_SG_POLLING*. The first calls a function that using the ID of the devices gets the ID of its interrupts and then, if it was defined in the config, it is injected into the service guest. The second event, calls a function that pushes the ID of the device into the list *pollingdevicelist*.

At this moment, the execution goes back to the service guest. If the VirtIO uses interrupts, when the interrupt associated with the device is injected, it is incremented a flag and it will be called a function

named **type of device*_mmio_callback_handler*. If the device does not use interrupts, from time to time it will call a function that uses a hypercall with the operation *POLLING_OP (3)* to check if there was any access to a MMIO register.

To simplify the use of the hypercall, it was created the *virtio_hypercall*, which has as parameters the ID of the device, the offset of the register, the operation and the value. It returns all these values plus the width of the access. This way, it can support every hypercall use that is required.

In what concerns polling, the use of the *virtio_hypercall* is simple. The ID of the device is unknown, so the only parameter that is not zero is the operation.

```
virtio_hypercall(0, 0, POLLING_OP, 0);
```

If there is a new access that must be processed, the hypercall will return the ID of the device as well as the offset of the register that was accessed, the operation (write or read) and the value in the case of a writing. After that, it is called the function **type of device*_mmio_callback_handler*. Otherwise, it will get out of the function and repeat the procedure.

If the device uses interrupts, the first step is to use a hypercall to query Bao what has happened. This operation will give the necessary information to the service guest to process the access. This is done by using the operation *ASK_OP (2)*.

The use of the hypercall is presented below. It is necessary to send as a parameter the ID of the device in question and the operation *ASK_OP*. The other parameters must be zero.

```
virtio_hypercall(device.id, 0, ASK_OP, 0);
```

The return is the same as when is used the operation *POLLING_OP (3)*.

At this point, using interrupts or not, the procedure becomes the same. The service guest processes the values that were returned by the hypercall, executes the actions that are required and then, the service guest uses the hypercall, again, to send back the value of the register in case of a reading and to notify the driver guest in case of a writing. This time, the operation that is sent to the hypercall is *WRITE_OP (0)* or *READ_OP (1)*.

The use of the hypercall for these operations is demonstrated below.

```
virtio_hypercall(device.id, device.reg_off, device.op, device.value);
```

Now, moving on to the explanation of what can be the most crucial part of the mechanism, the VirtIO's hypercall. Every time the hypercall is called in the service guest, the execution changes to the hypervisor and it calls the function *hvc64_handler* that will read the type of hypercall and as this hypercall is a VirtIO hypercall, it redirects the execution to the function *virtio_hypercall* that is the handler to VirtIO hypercalls on Bao.

At the beginning of the function, Bao gets all the variables sent by parameter using the CPU's registers:

```
unsigned long ret = -HC_E_SUCCESS;
unsigned long dev_id = cpu.vcpu->regs->x[1];     // Device Id
unsigned long reg_off = cpu.vcpu->regs->x[2];    // MMIO register offset
unsigned long op = cpu.vcpu->regs->x[3];         // Operation
```

```
5 unsigned long value = cpu.vcpu->regs->x[4];      // Register value
```

After that, there is a switch case that sets the code that is executed according to the operation (*WRITE_OP*, *READ_OP*, *ASK_OP*, *POLLING_OP* or *INTERRUPT_OP*). If the operation that was sent as a parameter was *POLLING_OP*, it is checked if the ID of the device is zero and if that is true and there is at least one element in the list *pollingdevicelist*, it is removed an item from the list and it is updated the variable *dev_id* with the ID that was in the removed item. Thereafter, the execution continues to the section of the operation *ASK_OP*. Thus, when it is used the operation *POLLING_OP*, it is also used the operation *ASK_OP* consequently. In the section that belongs to the operation *ASK_OP*, it is confirmed both *reg_off* and *value* are zero. If this turns out to be false, then is returned a failure value. Otherwise, it looks for the device in the *virtiodevicelist* and then updates the values of the CPU registers that are used to send data from the hypervisor to the service guest, with the values of the list's element (*dev_id*, *reg_off*, *op*, *value* and *access_width*). In the case of the operation being both *WRITE_OP* or *READ_OP*, the procedure is the same. Firstly, it is called the function *virtio_hypercall_w_r_operation*. This function is responsible for the update of the device's parameter value of the list of devices (*virtiodevicelist*). If it returns false, it means that there was an error and the hypercall returns a failure value. If it returns true, it means that everything went as expected and then it is sent a CPU message to the CPU that originated the trap.

This CPU message is received in the function *virtio_handler* that redirects to the function *virtio_cpu_msg_handler*. This function is the final step. In the case of a writing, it will only wake the CPU, so that it can proceed. If the operation is a reading, it will write the value that was sent by the service guest to the register, so the driver guest can read the register's value and proceed with its execution.

3.3.2 Tests

After the integration of the VirtIO on the Bao hypervisor, the interface is almost ready to support any type of VirtIO device. For this reason, this section presents not only the tests of the mechanism implemented in the hypervisor, but also the final tests of the interface. These tests were conducted both in QEMU and in the Xilinx board.

Initially, Bao's VirtIO mechanism was tested. The different parts of the mechanism were tested individually and posteriorly, a test with all the components together was executed. Thereby, the first test purpose was simply to check the added configuration parameters. It consists of creating a configuration file, defining the parameters associated with the VirtIO and then printing their values on Bao's initialization to ensure they are correct.

Thereafter, it was tested the emulation of a VirtIO device. A simple code was created in the handler of the emulation for VirtIO devices that prints information of the access, including the value that was written in the case of a writing, the CPU register that was used and the length of the access that are essential for the mechanism. Every time the driver guest tries to access an MMIO register that belongs to a VirtIO device, there is a trap to Bao and it calls this function that allows the tracking of the access values. After this simple test it was executed a new test but this time, the goal was not to verify if the access was done

in the right way, but to check if Bao can update the register that the driver guest tried to access. Thus, at the end of the handler, it is called a function that writes a value to the CPU register that was used by the driver guest. Then, the driver guest prints the value of the register to make sure that the register's value was updated.

The final test in what concerns the emulation stage was to put the CPU in idle state. The most important part of this test is to guarantee that the values of all CPU registers are the same the moment before and after the idle. If this turns out to be false, it can produce massive chaos in the driver guest.

The next step was to test the communication between CPUs using the *cpu_msg* and the communication between the service guest and Bao through the VirtIO hypercall. The test of the *cpu_msg* was to create a CPU message, call the function *cpu_msg* and print a message in the handler of the other CPU to verify that the communication was well succeeded. The hypercall test consisted of multiple calls using different arguments and confirming that its return was correct.

The final test was the verification of the entire mechanism. With this in mind, it was created a test that could verify all the stages from the virtqueues, to the handle of accesses to MMIO registers, or hypercalls. Hence, in this test, the driver guest starts by initializing two virtqueues (transmit virtqueue and receive virtqueue) using the correspondent MMIO registers. This stage allows the handle of MMIO registers' verification as well as the verification of the service guest's mechanism to deal with these accesses. Afterward, the driver guest creates a message and a descriptor to contain information about it. After updating the descriptor's data with the information and flags that describe the message, it stores the descriptor in the transmit virtqueue. Then, it notifies the service guest by writing the ID of the virtqueue in the register *QueueNotify*. The service guest receives an interrupt derived from the access to the register and starts processing the last descriptor. After the processing, the service guest puts the descriptor as used and uses a hypercall to notify the driver guest, which will receive a device event where it can process the descriptor. At this stage, both communication using virtqueues and the emulation of the device were tested. After verifying that the entire mechanism is working as expected, it is time to apply it to a specific device.

4. VirtIO Devices

In this chapter, it is explained the VirtIO devices and its implementation. The implementation of these devices needs to be used in cooperation with the VirtIO interface presented before and these drivers are the connectors between the VirtIO interface with the physical devices' device drivers.

The first section of this chapter is focused on the VirtIO-console that was implemented as a way to test if the VirtIO interface was actually working as supposed, since it is a device with a simple and straightforward implementation. Furthermore, there was already an implemented driver for the physical console device.

The second section presents the VirtIO-net which is the main goal of this dissertation. It has a lot of similarities with the VirtIO-console specially in the way that the accesses to MMIO registers that belong to VirtIO devices are handled. However, its implementation is more challenging and besides that, it was necessary to implement a new driver for the physical device which added some difficulty when comparing with the VirtIO-console implementation.

4.1 VirtIO-Console

VirtIO-Console is a straightforward device for data input and output. This device may have one or more ports and each port has a pair of virtqueues, one for input and the other for output, used to exchange information between the front-end and back-end drivers. The first port (port 0) is mandatory and it is only possible to use more ports if the *VIRTIO_CONSOLE_F_MULTIPORT* bit is set. Furthermore, it has a pair of control virtqueues that are responsible to exchange information about configuration changes.

Figure 4.1 presents the VirtIO-console architecture. To send data, the front-end driver enqueues outgoing characters into the transmitting virtqueue (1) which will be read by the back-end driver (2). To receive data, it stores empty descriptors into the receiving virtqueue (3). Then, the back-end driver uses these empty descriptors to store information about the information received (4). By accessing the receiving virtqueue, the front-end can read the data received (5).

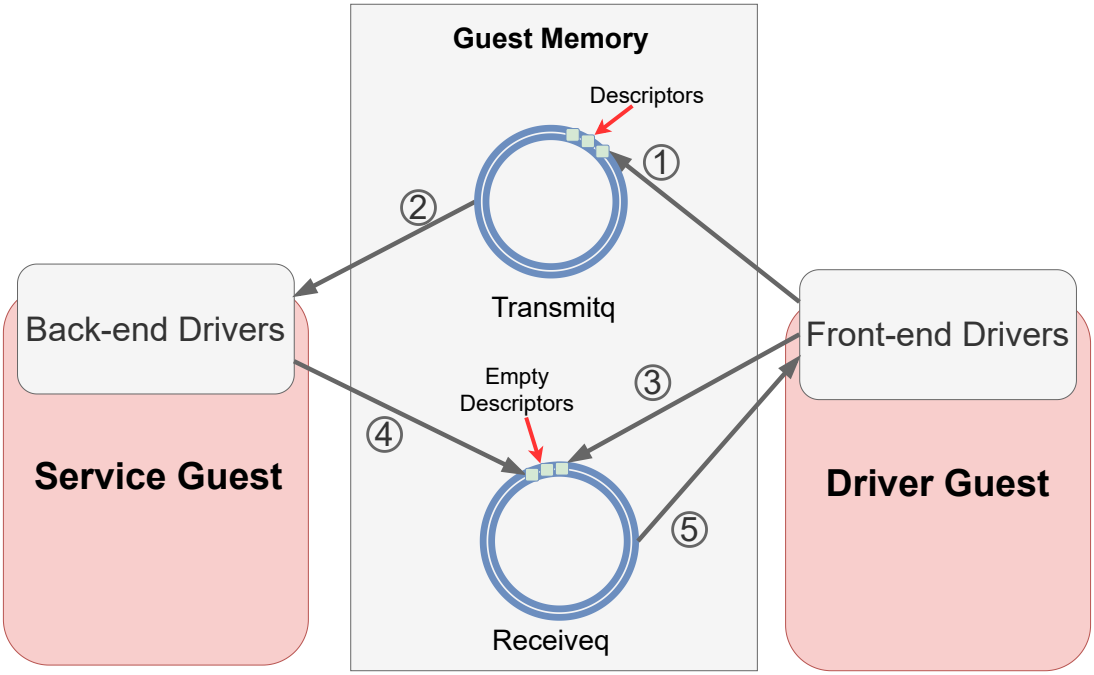


Figure 4.1: VirtIO-console.

This device can support three device-specific feature bits (*VIRTIO_CONSOLE_F_SIZE*, *VIRTIO_CONSOLE_F_MULTIPORT* and *VIRTIO_CONSOLE_F_EMERG_WRITE*). This back-end driver will not support any of these features, as its only purpose is to verify if the VirtIO interface can deal with a simple device.

The console device has the following variables that can be configured by using the config space:

- Number of columns;
- Number of rows;
- Number of the maximum number of ports;
- Emergency write.

However, they are only valid if the correspondent feature bits are set. The size of the console (number of columns and rows) is only supplied if the bit *VIRTIO_CONSOLE_F_SIZE* is set, the maximum number of ports can only be defined if the bit *VIRTIO_CONSOLE_F_MULTIPORT* is set. Otherwise, it is used only one port. Furthermore, if the feature bit *VIRTIO_CONSOLE_F_EMERG_WRITE* is set, the front-end driver can use the emergency write to send a single character without requiring the initialization of virtqueues.

4.1.1 Implementation

Initially, it was created the back-end driver to support the VirtIO-console device to allow testing the interface created with a real device. At first, only the front-end driver was created. By emulating the

interface with QEMU, it is possible to use its VirtIO-console back-end driver, which is ideal since it has a great level of compatibility. In the initial stage, the hypervisor was not used. It was only used a single bare-metal: the driver guest. This way, it was possible to create the first functional front-driver of the VirtIO-console.

The second step was to use the front-end (already tested with the back-end driver provided by QEMU) and implement a back-end driver for the VirtIO-console in the service guest. This stage required the integration of VirtIO with Bao, explained previously in Section 3.3.

The final step was to use the service guest with the VirtIO-console back-end driver with a Linux guest. This is the most important stage because Linux is truly popular in the embedded world and it is essential that the VirtIO interface is compatible with it.

For the implementation of the VirtIO-console back-end driver, it was created a structure to group all the information the VirtIO-console device needs. The *virtio_console* groups the virtqueues (two for data transmission and two for control purposes) and the *virtio_device* that contains all the data of the VirtIO device. This structure also incorporates the struct *config_space* that includes the values of the config space parameters:

```
struct virtio_console
{
    struct virtq receiveq;
    struct virtq transmitq;
    struct virtq control_receiveq;
    struct virtq control_transmitq;
    struct virtio_device device;
    struct virtio_console_config config_space;
};
```

Listing 4.1: VirtIO-console structure.

Despite not being used, as the feature bits that enable these parameters are not supported by this driver, the struct *virtio_console_config* was created to store the parameters of the config space. This struct is an essential part for the future work of this driver:

```
struct virtio_console_config
{
    uint16_t cols;
    uint16_t rows;
    uint32_t max_nr_ports;
    uint32_t emerg_wr;
};
```

Listing 4.2: VirtIO-console configuration space struture.

Several functions were implemented to handle the interrupts, one for each virtqueue:

```
static void virtio_console_receiveq_handler(void);
static void virtio_console_transmitq_handler(void);
```

```
3 static void virtio_console_control_receiveq_handler(void);
4 static void virtio_console_control_transmitq_handler(void);
```

Listing 4.3: VirtIO-console handlers.

Besides the functions that were already presented, it was implemented the function *virtio_console_mmio_callback_handler*. This is one of the most crucial functions of the VirtIO-console implementation, since it is in charge of reacting to the accesses of the MMIO registers that belong to this VirtIO device.

Every time there is a new access to a MMIO register, there is a trap to the hypervisor that will notify the service guest. Then, the service guest calls this function that starts by communicating with the hypervisor to get information about the access. After that, it needs to check if the access actually means something and if it was done in the correct way. For that, it checks if the width of the access is superior to four or equal to one or three, which indicates there was an invalid access. It checks if the width of the access is less than four and at the same time if the offset is less than 0x100 (for offsets lower than 0x100 all the accesses must be of four bytes and if this does not happen, the access is invalid). Finally, it checks if the offset of the register is greater than 0x200, which means that the access does not have any meaning because the MMIO registers are under the offset 0x100 and the config space ends before the offset 0x200. If any of these tests detect an error, the service guest stops the execution and sends a notification to the driver guest, so that the error can be perceived. Afterward, the execution is divided in two parts. The first, is where the config space accesses are handled. Here it is possible to receive two type of accesses: 16 and 32 bits, with register offsets higher than 0x100. The second part is where the rest of the MMIO registers, with an offset that is lower than 0x100, are handled. This second part is the most laborious due to the large amount of different operations that are necessary to be executed according to the register that was accessed.

The second part of the function *virtio_console_mmio_callback_handler* can also be divided into two. The first one is when the driver guest tries to read a register. Here, the only procedure that is done is reading the value of the register and then use a VirtIO hypercall to send the value to the hypervisor. However, there is one exception, the register *DeviceFeatures*. When the register *DeviceFeatures* is accessed, it is required to check the value of the register *DeviceFeaturesSel* to understand if the driver guest wants to read the 32 most significant bits or the 32 less significant bits. After that, it is sent the value of the 32 bits that the driver guest choose.

The second part of the function's second stage occurs when the driver guest tries to write to a register. When this happens, the service guest gets the value that the driver guest wants to write and updates the value of the register, unless it is part of the registers that require a special operation when accessed. The first register that has a special operation is the *DriverFeatures*. When the driver tries to write to this register, the service guest needs to compare the bits with the feature bits that it supports. If the subset sent by the front-end is contained in the feature bits the service guest contains, a variable named *verification* gets the value true, if not it is false. This variable is used when the driver guest wants to write to the register *Status*

the bit *FEATURES_OK*. This happens in the end of the feature negotiation. If the service guest did not support the subset of feature bits the driver sent, the negotiation was not well succeeded and the device initialization cannot proceed. So, if the variable *verification* is false and the driver guest tries to set the bit *FEATURES_OK* of the register *Status* the service guest does not allow it.

If the register *QueueNotify* was accessed, it is called the function to handle a new available descriptor. Then, there are the registers that associated to the virtqueues (*QueueSel*, *QueueNum*, *QueueDescLow* and *QueueDescHigh*). When a queue is selected by writing an ID into the register *QueueSel*, it is crutial to update instantly the registers *QueueReady* and *QueueNumMax*. When the register accessed is the *QueueNum*, it is necessary to change the value of the virtqueue's size according to the one that was chosen. In what concerns the access of the registers *QueueDescLow* and *QueueDescHigh*, it is obligatory to change not only the registers but also the value of the descriptor base of the virtqueue that was selected, taking into account that each of these registers represent only 32 bits off the physical address that is used in the service guest.

When it comes to the interacting with the physical device, it is called a uart function to put a character. Every time the register *QueueNotify* is accessed, meaning that there is a new available descriptor to handle, it is called the function *uart_putc* that takes a character and sends it to USART physical device, repeatedly until the end of the data. It is possible to know the end of the data by accessing to the descriptor's parameter *length*. To receive data from the physical device, it is used the device interrupt handler. When there is a uart interrupt, it means that the physical device receive a character. So, the interrupt handler gets the character by using the function *uart_irq_getchar* and stores it in an array. When the character is a '\0' which is a string terminator, it is activated a flag to send the string to the front-end. Afterward, the first available descriptor in the virtqueue is updated and the string is copied to address that is defined by the descriptor. Finally, the front-end is notified so that it can process the received data.

4.1.2 Tests

Initially, it was created the front-end driver. To test it, it was used the VirtIO-console back-end driver provided by QEMU. This test does not use Bao, it is simulated only a bare-metal. When the driver guest access a MMIO register that belongs to the VirtIO-console device, the QEMU's back-end takes control of the execution and handles the access. Two virtqueues were created and all the stages of the device initialization were tested. To verify if the device was working as expected, the driver guest created a message, stored it in the memory and created a descriptor to describe it. After putting it in the transmit virtqueue it wrote the virtqueue's ID into the register *QueueNotify* to notify the QEMU's back-end driver that there is a new buffer to be processed in the virtqueue. The back-driver processed the access, read the descriptor and then printed the message in the physical console device that is associated with the VirtIO-console device. Therefore, it was possible to check if the VirtIO-console front-end driver was well implemented by comparing the message printed with the message that was created in the driver guest.

After the first test, with the front-end driver already tested, the VirtIO-console back-end driver's implementation started. Before testing the back-end driver, it was indispensable to test the driver of the physical device, since the back-end driver calls its functions. This test and the following ones use the Bao hypervisor and they were executed in the Xilinx board as they do not have any QEMU's dependencies. After testing the functions responsible for sending and receiving data, it was tested the back-end driver implementation. To test the back-end driver, two bare-metal guests (service guest and driver guest) were run with the Bao hypervisor. The service guest incorporates the VirtIO-console back-end driver and the driver guest integrates the VirtIO-console front-end driver, previously implemented and tested. The test was similar with the one executed with the QEMU's back-end. However, it was printed more information such as the values of the descriptors and the values of all the MMIO registers accesses. It was essential to make sure that the device initialization and the feature negotiation did not present any error since it is a fundamental part that if it has a defect, it can lead to serious issues and a simple wrong bit on a register can make the device not operable.

After testing meticulously the back-end driver, the driver guest was replaced by a Linux guest, in order to perform the final test. The test with the Linux guest was a bit different. It was called the command *echo* in the Linux guest, to send a message to the VirtIO-console device. The Linux guest is responsible for dealing with this request and use the VirtIO mechanism to transmit the message to the service guest. Eventually, the service guest receives a notification and process the virtqueue. In the end, the service guest uses the physical console device to print the message.

The main obstacle faced on this final test was the initialization, because sometimes the linux guest received interrupts not related to VirtIO while the service guest was still processing the access to a MMIO register. This means that if the back-end is slow processing the access, the linux guest continues with is execution and the negotiation fails because it cannot get the correct register values. The mechanism used a flag to signalize that there was a new access and this flag was tested in a loop. This issue was solved simply by moving the back-end driver's code that was not directly in the handler of the interrupt that is generated by Bao when there is a new access to a MMIO register.

4.2 VirtIO-Net

VirtIO network device, also known as VirtIO-net, is a virtual ethernet card. This device is the most complex supported by VirtIO, but it is also one of the most useful, as it allows transmitting and receiving data from the equipment that is being used, to the network.

Similarly to the VirtIO-console device, it uses a virtqueue to transmit data and other to receive. Besides these two virtqueues, there is also a control queue, which only exists if the bit *VIRTIO_NET_F_CTRL_VQ* is set, used to control advanced filtering features. To transmit data, the outgoing packets are enqueued into the transmit queue. To receive data, it is necessary to place empty descriptors on the reception virtqueue. It is advisable to keep the reception queue as fully populated with empty descriptors as possible since if

the empty descriptors run out, the back-end driver cannot send new packets to the front-end driver which will lead to a drop in the performance.

Each network transmit buffer consists of a header followed by the network packet. The header incorporates ten parameters, but the last three (*hash_value*, *hash_report* and *padding_reserved*) only exist if the bit *VIRTIO_NET_F_HASH_REPORT* is negotiated. The back-end driver will not support these features, so they will not be explained. The other seven parameters are the following:

- flags: it can have one of four values: *VIRTIO_NET_HDR_F_NEEDS_CSUM* (1), *VALID* (2), *INFO* (4) or zero;
- gso_type: if the bit *VIRTIO_NET_F_HOST_TSO4*, *TSO6* or *UFO* was negotiated and the packet expects TCP segmentation or User Datagram Protocol (UDP) fragmentation this parameter is set to *VIRTIO_NET_HDR_GSO_TCPV4*, *TCPV6* or *UDP* respectively. Otherwise, this parameter is set to *VIRTIO_NET_HDR_GSO_NONE*. If this happens it is possible to transmit packets larger than 1514 bytes;
- hdr_len: it contains the size of the header;
- gso_size: it carries the size of the packet removing the header;
- csum_start: it controls the offset of the packet where the checksumming will begin;
- csum_offset: it indicates how many bytes after the *csum_start*, the new checksum will be;
- num_buffers: it indicates how many descriptors are used to describe the packet. This allows the packet to be spread over multiple descriptors. If the value of *num_buffers* is one or the feature bit *VIRTIO_NET_F_MRG_RXBUF* was not negotiated, it means that the packet will not be divided and it is contained in a single buffer that follows the header.

This device can support twenty-eight device-specific feature bits, which won't be explained due to the large number of features and because the VirtIO-net back-end driver won't be compatible with any of them.

The device configuration field incorporates the following nine parameters that can be changed by accessing the config space:

- Media Access Control (MAC) address: always exists, but only is valid if the feature bit *VIRTIO_NET_F_MAC* is set;
- Status: can have the value *VIRTIO_NET_S_LINK_UP* or *VIRTIO_NET_S_ANNOUNCE* and only exits if the feature bit *VIRTIO_NET_F_STATUS* is set.
- Maximum number of virtqueue pairs: specifies the maximum number of virtqueue pairs (transmit and receive virtqueues) that can be configured. This parameter only exists if *VIRTIO_NET_F_MQ* or *VIRTIO_NET_F_RSS* is set. Otherwise, the maximum number of virtqueue pairs is one;

- Mtu: stipulates the maximum MTU (Maximum transmission unit) for the front-end driver to use and only exists if the bit *VIRTIO_NET_F_MTU* is set;
- Speed: controls the device speed (MBit/s) and only exists if *VIRTIO_NET_F_SPEED_DUPLEX* is set;
- Duplex: only exist if *VIRTIO_NET_F_SPEED_DUPLEX* is set and it is used to choose between full duplex, half duplex or unknown duplex state;
- Receive Side Scaling (RSS) maximum key size: specifies the maximum supported length of the RSS key (bytes). It only exists if the feature bit *VIRTIO_NET_F_RSS* or *VIRTIO_NET_F_HASH_REPORT* is set;
- RSS maximum indirection table length: stipulates the maximum number of 16-bit entries in RSS indirection table and it is used only if the bit *VIRTIO_NET_F_RSS* is set;
- Supported Hash Types: only exists if the bit *VIRTIO_NET_F_RSS* or *VIRTIO_NET_F_HASH_REPORT* which means that the device supports hash calculation. This parameter allows specifying the hash type to be applied [24].

Besides the VirtIO-net back-end driver, it was also necessary to create a device driver for the physical device. To do so, it was used a Xilinx's driver, the XEmacPs. This device driver manages configuration and control, as well as the sending and receiving of ethernet frames. Even though the settings of many devices are greatly different, a single device driver can handle them all. A single XEmacPs device driver can handle multiple devices even if they present significant differences in their configuration.

It uses buffer descriptors to describe ethernet frames and Direct memory access (DMA) to store them. Each buffer descriptor describes the memory region that contains a full or a partial ethernet packet. These buffer descriptors are usually in a list, which the hardware follows when sending or receiving packet buffers.

4.2.1 Implementation

Unlike the VirtIO-console's implementation, it was not used the same procedure to create the back-end, because it is harder to create a functional front-end for this device. Thus, it was created the back-end driver right from the beginning using the front-end driver from a Linux guest. With this in mind, it was created a struct to the VirtIO-net device that contains the virtqueues to the transmission and reception as well as the control virtqueue. It also incorporates the struct that contains the information about the device:

```
1 struct virtio_net {
2     struct virtq receiveq;
3     struct virtq transmitq;
4     struct virtio_net_ctrl controlq;
5     struct virtio_device device;
```

```
};
```

Listing 4.4: VirtIO-net structure.

The control virtqueue functioning was not implemented because the back-end driver does not support the feature bits that enable the utilization of its parameters. Nevertheless, its struct was created, so it can be used in a future implementation, as it is an essential component of a VirtIO-net device back-end driver with full compatibility. This virtqueue is a bit different from the other virtqueues, so it was necessary the creation of a new struct (*virtio_net_ctrl*) to incorporate its parameters. The first three parameters of this structure (*class*, *command* and *command_specific_data*) are set by the front-end driver whilst the last parameter (*ack*) is set by the back-end driver.

```
struct virtio_net_ctrl {
    uint8_t class;
    uint8_t command;
    uint8_t command_specific_data[16];
    uint8_t ack;
};
```

Listing 4.5: VirtIO-net control virtqueue structure.

As explained in Section 4.2, the configuration space is composed of nine parameters that can be accessed by the front-end driver by using the MMIO registers with offsets after 0x100. To group all the parameters of the config space, it was used the struct *virtio_net_config*. To set these parameters, the front-end must access the address with an offset of 0x100 to change the first value of the MAC address. The following five bytes define the other five values of the MAC address. Then the address with an offset of 0x106 is used to access the status. To access the following parameters, it is only necessary to add its size (in bytes) to the offset for each.

```
struct virtio_net_config {
    uint8_t mac[6];
    uint16_t status;
    uint16_t max_virtqueue_pairs;
    uint16_t mtu;
    uint32_t speed;
    uint8_t duplex;
    uint8_t rss_max_key_size;
    uint16_t rss_max_indirection_table_length;
    uint32_t supported_hash_types;
};
```

Listing 4.6: VirtIO-net configuration space structure.

An essential part of the VirtIO-net packet communication is the header. The struct *virtio_net_hdr* was created to contain all the parameters that are essential to accommodate the information of each packet.

```
struct virtio_net_hdr {
```

```
    uint8_t flags;
    uint8_t gso_type;
    uint16_t hdr_len;
    uint16_t gso_size;
    uint16_t csum_start;
    uint16_t csum_offset;
    uint16_t num_buffers;
};
```

Listing 4.7: VirtIO-net header structure.

Besides these structs, it was also necessary to create functions to handle the interrupts. Each virtqueue has its own handle because each one of them has a different reaction to an interrupt.

```
void virtio_net_receiveq_callback_handler(struct virtio_net *);
void virtio_net_transmitq_callback_handler(struct virtio_net *);
void virtio_net_controlq_callback_handler(struct virtio_net *);
```

Listing 4.8: VirtIO-net handlers.

Finally, the most essential function of the VirtIO-net implementation: the *virtio_net_mmio_callback_handler*, which is in charge of handling all the accesses to MMIO registers that belong to the VirtIO-net device.

This function is similar to the function *virtio_console_mmio_callback_handler* explained in Section 4.1.1 that has the same role but for the VirtIO-console device. The main difference is in the section where is handled the access to the config space. This is the MMIO region that belongs to the device after the offset 0x100. The config space from the offset 0x0 to 0x100 is exactly the same, as they are not device-specific. That is why both VirtIO-console and VirtIO-net seem a lot like each other. However, the config space is device specific, which means that the data that will be in the offset 0x100 of a VirtIO-console is completely different from the data that will be in the same region of a VirtIO-net device or another VirtIO device. For instance, the VirtIO-console uses four variables to configure the device, two with 16 bits (2 bytes) and the other two with 32 bits (4 bytes), which means that the config space ends in the offset 0x10B. The VirtIO-net device uses nine parameters that can be of 8, 16 or 32 bits and its config space ends in the offset 0x117. With this in mind and since the console only supports parameters of two or four bytes, it was indispensable to allow and handle accesses of a single byte in the config space.

4.2.2 Tests

The front-end driver of a VirtIO-net device is a lot more complex than the front-end driver of a VirtIO-console device. For that reason, it was not implemented the front-end of the VirtIO-net device in a bare-metal VM. Instead, it was used a Linux guest which contains a VirtIO-net front-end already implemented.

Initially, it was essential to test the Xilinx device driver. Xilinx provides a simple example that transmits data and waits for its echo using physical loopback. In the end, it compares the message received to the

one that was transmitted to verify if they are the same. Afterward, it was necessary to create an adaptation of this code to be possible to use with the VirtIO mechanism previously created. This adaptation was tested using a single bare-metal running over the Bao hypervisor. As it uses physical loopback the only test that is logical is verifying if the data that was transmitted is the same as the one that was received. With the implemented device driver is not possible to send data to the exterior. Consequently, it is not possible to use a network protocol analyzer such as Wireshark to capture the packets that were sent and analyze its content. This device driver was very problematic and it was necessary to disable the cache in the baremetal, since it required a lot of cache operations. Moreover, the buffer descriptors used by the device driver need to be allocated in uncached memory. After this fix it was possible to move to the next test.

After guaranteeing that the device driver was working as expected, it was tested the VirtIO-net itself. Initially, the device driver was not used and it was only tested the initialization of the device. It was crucial to make sure that the Linux guest could find the VirtIO-net device. After making sure Linux could discover the device, it was tested if it could initialize it, without any error. A crucial test was the feature negotiation because if the feature negotiation fails it is impossible to use the device to communicate. Then, it was verified if the VirtIO-net front-end could receive the packets that the Linux guest sent. It was crucial to verify not only the data, but also the values of all header's parameters.

Afterward, it was time to test the final implementation with requests from the Linux guest being handled by the service guest that calls the device driver functions to send and receive data from the device. This test was well succeeded. However, the device driver was not capable of sending the packets to the exterior because it uses physical loopback. This means that the responses to the Linux guest requests are the requests themselves, because the input of the physical is equal to the output. This is not appropriate. Despite the main objective of this dissertation (create a VirtIO interface for Bao and a back-end for the VirtIO-net) working, the device driver does not follow the same path as it was not possible to implement it without using physical loopback, which means that it cannot communicate with the exterior world. This issue in the device driver does not allow a complete test of the VirtIO-net back-end.

All the VirtIO-net device tests were performed on the zcu-104 board.

5. Conclusion

The use of embedded systems has increased in the last years and so has their complexity. The use of multicore system has become a must since it allows running applications with different criticality levels on the same hardware platform. This can be risky as they can interfere with each other. The virtualization arrived to fix this issue. By using a hypervisor, it is possible to isolate the operating systems both spatially and temporally. However, most hypervisors are not prepared for the embedded world and that is why the Bao hypervisor emerged. In what concerns peripherals, this hypervisor only gives pass-through access, not being possible to share devices between different guests. This dissertation solves this issue by using VirtIO.

In the first stage of this dissertation, it is designed and the implemented the interface that allows two or more guest to communicate with each other. It all started by creating a mechanism with simple virtqueues that were stored in shared memories. To notify the opposite guest that there is a new message, Bao's hypercalls that are associated with the shared memories were used. After that, it was implemented the VirtIO interface using the packed virtqueue layout, the most recent and less problematic virtqueue layout. Then, it was designed and implemented the integration with Bao hypervisor, one of the most crucial parts of this work. It was necessary to change Bao's code to allow handling accesses to MMIO registers that belong to VirtIO devices.

The second stage was focused on the design and implementation of two VirtIO devices back-end drivers: VirtIO-console and VirtIO-net. It was created a simple and functional back-end driver for each of these devices by only implementing the support for the crucial features. Initially, the back-end driver for the VirtIO-console device was created and it was integrated with the already existent device driver of the console physical device. This stage's goal was to verify that the VirtIO interface was truly functional and it was possible to share a physical device using it. The implementation of the VirtIO-console back-end served as a basis to the VirtIO-net back-end implementation, which was the main purpose of the second stage of this dissertation. After the implementation of the VirtIO-console back-end it was implemented the VirtIO-net back-end as well as a device driver for the physical device as it was non-existent. The only purpose of this device driver was to execute a complete test of the device's virtualization, which turned out to be not feasible.

The Bao hypervisor does not allow a VM to access to the memory of another VM guaranteeing full isolation of the memory. The only way to share memory between two guests is by creating a shared

memory region. Using Linux's front-end drivers of VirtIO devices, it was impossible to guarantee that the descriptors will be placed in the shared memory. Thereby, it was necessary to change the Bao's code to allow the service guest to access to the driver guest's memory. This way, it could access all the descriptors as well as the data that they describe without any issues. This is not ideal, since it is lost one of the Bao's main goals: isolation between guests.

5.1 Future Work

Despite the fact that the VirtIO interface is functional, which was the main goal of this dissertation, there are a lot of improvements that would be interesting.

It was impossible to execute a complete test of the virtualization of the net device, due to problems in the device driver. Bearing that in mind, the device driver should be changed in such a way that allows the communication with the exterior world by not using physical loopback.

The implementation of the split virtqueue layout. Although the packet virtqueue layout is less susceptible to issues such as bad cache utilization or bad performance and is the most recent layout, which means that the recent systems already support it, there are still a lot of systems that are not compatible with this virtqueue layout. Thereby, it would be interesting to add the split virtqueue to this dissertation's VirtIO interface to make it compatible with older systems that only support the split virtqueue.

In this dissertation, it was implemented two VirtIO devices: VirtIO-console and VirtIO-net. However, version 1.1 of the VirtIO's specification already specifies nineteen devices. Even though most of them are not really useful for Bao, it would be interesting to implement more drivers for VirtIO devices to amplify the number of supported devices.

Another essential improvement concerning the compatibility of the interface is the support of feature bits. Both VirtIO-console and VirtIO-net were created with the goal of being simple and functional. For this reason, it was focused on the functioning and not on the full compatibility of the system. Thus, to improve the compatibility of both VirtIO-console and VirtIO-net it would be essential to add support to the device-specific feature bits.

The service guest was implemented in a bare-metal guest. With the increase of VirtIO devices being used it would be better to opt for a multi-thread/multi-task guest such as FreeRTOS, a Real-time operating system for microcontrollers. With this implementation, it would be possible to deal with multiple accesses of different devices at the same time, since they would be divided into multiple tasks, which could lead to better performance.

Another important future work is the evaluation of the performance. It is well known that the use of VirtIO leads to worse performance than the use of pass-through devices. This is the price to pay for sharing devices. It would be interesting to test the execution times of both, so it would be easier to do trade-offs between the use of pass-through or VirtIO.

Moreover, it would be essential to change the VirtIO's mechanism in order to be possible to use shared memories both for the storage of descriptors and the data storage. Thus, it would be possible to use the original Bao's code with full memory isolation.

In what concerns the VirtIO-net back-end driver, it would be interesting to create features such as the possibility of multiplexing the physical device. In other words, create a mechanism that can use different IP addresses for different guests but uses the same physical device. By creating a sort of network bridge, it would be possible to use a single ethernet card by two guests completely isolated.

References

[1] J. Martins and S. Pinto, "Bao: a modern lightweight embedded hypervisor," *Embedded World 2020 Exibition and Conference*, 02 2020.

[2] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded Hypervisor Xvisor: A Comparative Analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 682–691, 2015.

[3] G. Heiser, "Virtualizing embedded systems - why bother?," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 901–905, 2011.

[4] R. Kaiser, "Complex embedded systems - A case for virtualization," in *2009 Seventh Workshop on Intelligent solutions in Embedded Systems*, pp. 135–140, 2009.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, p. 164–177, oct 2003.

[6] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," *Workshop on Next Generation Real-Time Embedded Systems*, 01 2020.

[7] "ARM Cortex-A Series Programmer's Guide for ARMv8-A." `https://developer.arm.com/documentation/den0024/a/Fundamentals-of-ARMv8`. Accessed: 2022–10-21.

[8] A. Iqbal, N. Sadeque, and R. I. Mutia, "An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems," *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2022.

[9] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?," in *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, pp. 1–7, 2010.

[10] M. Rosenblum, "The Reincarnation of Virtual Machines: Virtualization Makes a Comeback.," *Queue*, vol. 2, p. 34–40, jul 2004.

[11] G. Heiser, "The Role of Virtualization in Embedded Systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, (New York, NY, USA), p. 11–16, Association for Computing Machinery, 2008.

[12] G. Heiser, "Secure embedded systems need microkernels," *USENIX*, vol. 30, 01 2006.

[13] J. E. Smith and R. Nair, "Chapter Eight - System Virtual Machines," in *Virtual Machines* (J. E. Smith and R. Nair, eds.), The Morgan Kaufmann Series in Computer Architecture and Design, pp. 369–443, Burlington: Morgan Kaufmann, 2005.

[14] J. Shropshire, "Analysis of Monolithic and Microkernel Architectures: Towards Secure Hypervisor Design," in *2014 47th Hawaii International Conference on System Sciences*, pp. 5008–5017, 2014.

[15] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors," in *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems*, APSys '10, (New York, NY, USA), p. 19–24, Association for Computing Machinery, 2010.

[16] "Virtio devices high-level design." `https://projectacrn.github.io/2.1/developer-guides/hld/hld-virtio-devices.html?fbclid=IwAR3H5Ra50_yrLOqtKW9X7s5QNOvJ9VwHOSf28LHr1APsS0IubbqlTj1GVq0`, 2020. Accessed: 2021–12-29.

[17] B. Sá, J. Martins, and S. Pinto, "A First Look at RISC-V Virtualization from an Embedded Systems Perspective," *IEEE Transactions on Computers*, 03 2021.

[18] F. Diakhaté, M. Pérache, R. Namyst, and H. Jourdren, "Efficient shared memory message passing for inter-VM communications," *hal-00368622*, 2008.

[19] J.-H. Kim and H.-W. Jin, "Virtio Front-End Network Driver for RTEMS Operating System," *IEEE Embedded Systems Letters*, vol. 12, no. 3, pp. 91–94, 2020.

[20] J. Durand Wesolowsk, A. Boudguiga, A. Patel, J. Viard De Galbert, M. Donain, W. Klaudel, and G. Scigala, "Xvisor VirtIO-CAN: Fast Virtualized CAN," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, (TOULOUSE, France), Jan. 2016.

[21] S. Patni, J. George, P. Lahoti, and J. Abraham, "A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial," in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pp. 6–9, 2015.

[22] D. Milea, "Hypervisor-less virtio," *Assembling Multi-OS systems using standards-based protocols for intra-SoC connectivity and device sharing*.

[23] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A Real-Time virtio-based Framework for Predictable Inter-VM Communication," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 27–40, 2021.

[24] M. S. Tsirkin and C. Huck, "Virtual I/O Device (VIRTIO) Version 1.1, OASIS Committee Specification." `https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html`, 2018.

[25] E. P. Martín, "Virtio devices and drivers overview: The headjack and the phone." `https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone`, 2020. Accessed:

2021–10-20.

[26] R. Russell, "Virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, 2008.

[27] E. P. Martín, "Virtqueues and virtio ring: How the data travels." `https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels`, 2020. Accessed: 2021–10-20.

[28] E. P. Martín, "Packed virtqueue: How to reduce overhead with virtio." `https://www.redhat.com/en/blog/packed-virtqueue-how-reduce-overhead-virtio`, 2020. Accessed: 2021–10-20.

[29] A. Adam, A. Ilan, and T. Nadeau, "Packed virtqueue: How to reduce overhead with virtio." `https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net`, 2019. Accessed: 2021–10-20.

[30] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto, "TZ- VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor," *IEEE 27th International Symposium on Industrial Electronics (ISIE), Cairns, QLD, Australia*, 06 2018.

[31] E. P. Martín, "Deep dive into Virtio-networking and vhost-net." `https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net`, 2019. Accessed: 2021–10-20.

[32] J. D. Pagare, N. A. Koli, and G. Baba, "A technical review on comparison of Xen and KVM hypervisors: An analysis of virtualization technologies," *International Journal of Advanced Research in Computer and Communication Engineering*, pp. 8828–8832, 2014.

[33] Santos, Jose Renato and Turner, Yoshio and Janakiraman, G. and Pratt, Ian, "Bridging the gap between software and hardware techniques for i/o virtualization," in *USENIX 2008 Annual Technical Conference*, ATC'08, (USA), p. 29–42, USENIX Association, 2008.