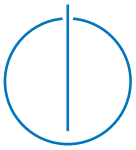
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Rethinking IO emulation architectures for VMs

Sandro-Alessio Gierens



DEPARTMENT OF INFORMATICS

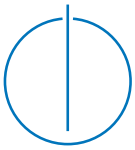
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Rethinking IO emulation architectures for VMs

Überdenken von IO Emulationsarchitekturen für VMs

Author:	Sandro-Alessio Gierens
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Peter Okelmann, M.Sc.
Submission Date:	15.09.2022



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2022 Sandro-Alessio Gierens

Acknowledgments

I would like to express my gratitude to my advisor, Peter Okelmann, for his support and guidance throughout the course of this thesis. Our weekly meetings kept the project on track and gave me most valuable ideas when I was stuck.

I am also thankful to the chair of Decentralized Systems Engineering for giving me the opportunity to work on such an interesting topic. It taught me a lot about the inner workings of virtualization and helped me to shape my academic journey further in the direction of systems programming.

Abstract

System virtualization has become an irreplaceable cornerstone of modern IT infrastructures. While CPU and memory virtualization excel through ubiquitous hardware support from processors, I/O virtualization is still often unable to keep up. In the absence of special solutions like PCI passthrough or SR-IOV (Single-Root I/O Virtualization), para-virtualization is the pre-dominant approach for virtual I/O devices. Virtualizing MMIO (Memory Mapped I/O) accesses to those devices however involves expensive operations, like context switches, causing overheads. The recently proposed KVM (Kernel-based Virtual Machine) feature, ioregionfd, therefore aims to circumvent those overheads with a file-descriptor-based communication between KVM and the VMM (Virtual Machine Monitor) in user space, in a more general fashion than previous optimizations. In this work we first give a comprehensive overview of the current state of MMIO virtualization in QEMU/KVM, necessary to understand the promise of ioregionfd. We then provide a proof of concept implementation applying it to QEMU's VirtIO MMIO device frontend. We demonstrate how we made it generic enough to be applied to other devices, and add best practices for debugging host kernel-based virtualization features like ioregionfd. Finally, we thoroughly evaluate the performance of our implementation in an effort to quantify the benefits of ioregionfd. Our evaluation on the VirtIO MMIO network device shows that it suffers from random packet loss by default, and there is more loss with ioregionfd. Whether ioregionfd causes packet loss on its own or merely amplifies a pre-existing issue is unclear, because the root cause of the packet loss has yet to be found, making our verdict about the benefit of ioregionfd inconclusive.

Contents

1. Introduction

System virtualization is not only an essential part of modern data center operation [1–5] , but also one of the technological pillars on which a big part of cloud computing rests [6–9] . A lot of effort has been put into tightening the performance gap between virtualized and native systems over the past decades [10–18] . Especially CPU and memory virtualization profit from hardware virtualization extensions built into modern processors [19–28] . Their combination of simple superiority over software alternatives and general availability has made them a de-facto standard for virtualized systems [29–35] . Similar approaches exist for I/O virtualization, like PCI passthrough and SR-IOV, but they often lack in flexibility, are complicated to setup or simply require expensive hardware [36–44] . Also cloud providers usually treat these as opt-in features and charge extra for them, while software-based I/O virtualization continues to be the most common solution [45–47] . Emulated or para-virtualized I/O devices operate with the same mechanisms as their physical counterparts [48–54] . They rely heavily on asynchronous I/O for their data planes, and can harness the easy yet efficient virtualization of DMA via shared memory [20, 48, 49, 55–60] . They also use optimizations like irqfd and ioeventfd for their interrupts and notifications, and with that can achieve remarkable throughputs [11–13] . Synchronous I/O, on the other hand, mostly found in their control plane but not exclusively, often suffers from expensive operations like CPU mode and context switches in the current implementation of guest PIO, partially being the cause for significant latency jumps with increasing I/O operation rate [14, 61–68] . With the growing demand for deploying virtual machines in I/O intensive applications, this becomes even more problematic [69–73] . Optimizations exist like ioeventfd, but they are only applicable to a part of the PIO operations [12, 13] . This is why kernel developers recently proposed a new mechanism called ioregionfd, which similar to ioeventfd allows to attach a direct file-descriptor-based communication channel between KVM and the hypervisor to guest's I/O memory regions. It distinguishes itself by defining a wire protocol for the channel, that offers more general operations than previous solutions [14, 74–76] . Due to its novelty only one patchset for I/O device emulators has been implemented so far. It uses ioregionfd to counter performance penalties of additional IPC that the security-focused isolation of the QEMU remote device incurs [77–80] . While the results look promising [79] , this does not answer the question of how viable ioregionfd is as a general solution for reducing guest PIO latencies.

In this thesis we therefore analyze the cause for potentially low guest PIO performance and understand how ioregionfd attempts to tackle it, to realize a second proof-of-concept implementation for the much more popular VirtIO device family. We also show how we made our code generic enough to suffice different devices with a single core handler

function. We furthermore provide leads on how to thoroughly debug host-kernel based features like ioregionfd and their communication with the VMM. Finally we evaluate the efficiency of our implementation with the high performance software packet generator MoonGen to better estimate the potential impact of ioregionfd on guest PIO latencies.

The remainder of this work is structured as follows: Chapter 2 provides the technical foundation necessary to understand the rest of the thesis, it first explains how CPU device communication is realized in bare-metal systems, before expanding the scope to virtualized systems with focus on QEMU/KVM guests. Chapter 3 then analyzes the impact of this on guest I/O performance, and more specifically first comparing why MMIO is much more problematic than DMA, before presenting the existing optimizations as well as the newly suggested ioregionfd. Chapter 4 continues by dissecting how ioregionfd could improve guest PIO and how this is used in the existing QEMU patchset, before exploring what other devices might profit from it. It then explains what VirtIO is and why we chose it for our proof-of-concept implementation and finishes with a set of further goals we have for our code. This leads over to chapter 5 which thoroughly explains how we realize those goals. It starts by explaining the process of mating the VirtIO device initialization with the ioregionfd setup, before focussing onto the wire protocol and its debugging. After that it shows how we divide the handler function for QEMU's end of the channel in a way that allows to easily extend the code to other devices with minimal changes. In the end the chapter explains the intricacies of the VirtIO MMIO bus and what registers we end up with applying ioregionfd to for our evaluation. Chapter 6 then takes over by establishing latency as the key performance indicator, describing how it is measured in general before presenting our test setup for comparing our implementation with the existing PIO handler, as well as with the native system. Then follow our concrete measurements and results before we end the thesis with discussing related work in chapter 7 and finally concluding it in chapter 8.

2. Background

This chapter will introduce the technical background necessary for understanding the remainder of this work. First it will describe how the central processing unit of a modern baremetal machine communicates by the means of direct memory access and programmed I/O with the systems I/O devices. Then it will illustrate how the QEMU hypervisor interoperates with KVM to enable device access for virtual machines.

2.1. Baremetal Device Access

When a core of the central processing unit (CPU) wants to access data it does not have in its registers, it will generally speaking command the respective controller within the CPU over the control bus to fetch the data, pointed to by the address it put on the address bus, to the data bus, see figure 2.1. [59, 81]

2.1.1. Memory Access

In case the core wants to access data within the system's main memory, it will talk to the memory controller (MC). In modern CPUs it usually combines multiple functions that were spread over different chips in older systems and is therefore nowadays sometimes referred to as unified memory controller (UMC). It contains the memory management unit (MMU) to translate virtual addresses to physical addresses with the help of the translation lookaside buffer (TLB). It also restricts the access a particular process has to the memory. After it found the physical address it translates it to the channel number, to select between the different DIMMs (dual in-line memory modules), and also row, bank and column numbers, to access the correct location within one of its DRAM (dynamic random memory access) cells. It then makes the access over the memory bus. [59, 60, 82, 83]

2.1.2. I/O Device Access

When the CPU and an I/O (input/output) device want to communicate, there are generally three ways this can be done, with interrupt I/O, direct memory access (DMA) or programmed I/O (PIO).

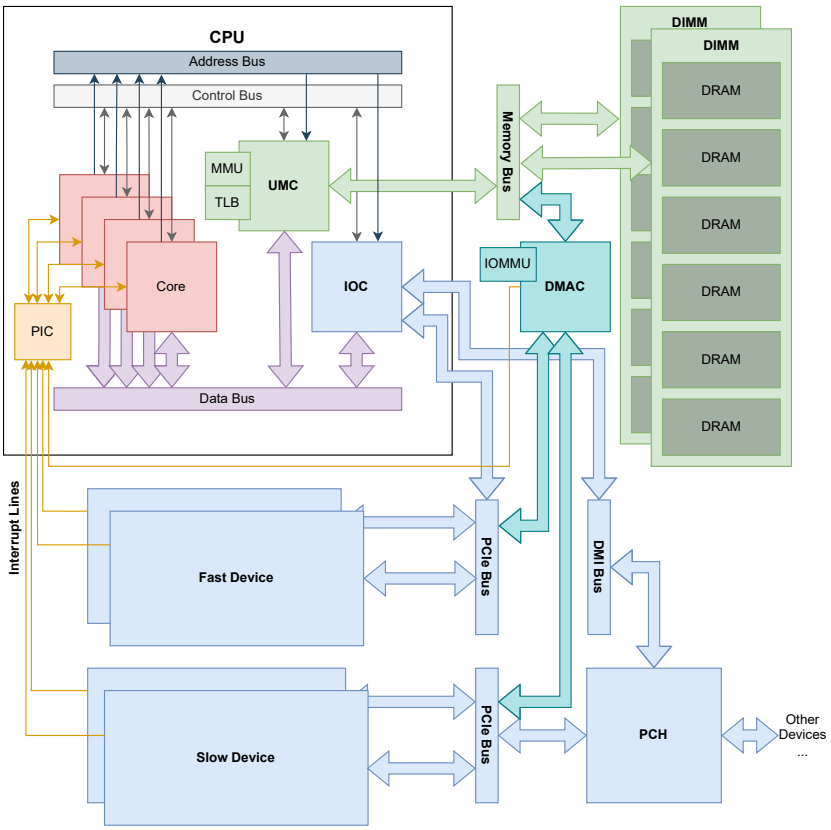


Figure 2.1.: Block diagram of CPU and motherboard components relevant for memory and I/O device access. The CPU (central processing unit) has several cores executing instructions. Accesses to outside data are handled over a set of busses (address, data, control bus). The UMC (unified memory controller) is responsible for accesses to main memory, and contains the MMU (memory management unit) and TLB (translation lookaside buffer) for address translation. The IOC (I/O controller) is responsible for accesses to I/O devices. The DIMMs (dual in-line memory modules) containing the DRAM (dynamic random access memory) cells are connected to the UMC over the memory bus. I/O devices are connected either directly to the IOC over a PCI (Peripheral Component Interconnect) bus, or indirectly over the PCH (platform controller hub) that is connected to the IOC over the DMI (direct media interface) bus. PIO (programmed I/O) goes over those busses. For DMA (direct memory access) the device busses are also connected to the DMAC (direct memory access controller) that contains the IOMMU (I/O memory management unit) for address translation. Both, devices and the DMAC, can notify the CPU about completion of a DMA operation, for example by sending an interrupt over the interrupt lines to the PIC (programmable interrupt controller). [58–60, 82–84] Based on [85–88]

2.1.2.1. Interrupt I/O

When an I/O device tells the CPU to stop its execution, to run a certain interrupt handler, this is called interrupt I/O. The device will send a so-called interrupt request (IRQ) to the CPU over an interrupt line. Apart from its use during DMA, interrupt I/O is for accesses from the device to the CPU. This work however focusses on the other direction, so we will not go into more detail here but instead focus on DMA, which although invoked by the device opens up a bidirectional communication path and PIO which is invoked by the CPU. [58–60]

2.1.2.2. Direct Memory Access

In contrast to the main memory, I/O devices are usually rather slow both in terms of throughput, but especially regarding latency, so the time they need to answer a request. Waiting for this is waist of the CPUs precious time, even more so the larger the amount of data involved is. The idea behind DMA therefore is, to give I/O devices direct access to the main memory without involving the CPU, so both CPU and device can work on a shared memory region asynchronously. [58–60]

Take a network card for example: It will use DMA to share its receive (RX) and transmit (TX) buffers with the CPU. When the CPU wants to send a packet it will simply write it into the TX buffer in memory and then continue its work. The network card will transmit the packet once its ready, without the CPU needing to wait for that. This becomes even clearer when looking at packet reception. Instead of actively trying to poll data from the network card that is not there yet, the process will be put to sleep while the CPU executes other tasks. When the network card receives the next packet, it simply writes it in the DMA RX buffer and then signals the CPU over its interrupt line, that there is data it might want to work on. [58–60, 89, 90]

Similar to the CPU accessing the memory over the UMC, I/O devices interact with the direct memory access controller (DMAC) over their respective bus. With the help of the I/O memory management unit (IOMMU) the DMAC makes sure the devices only access memory they are supposed to. If valid the DMAC translates the memory address to channel, row, bank and column numbers to access the right DRAM location over the memory bus. [58–60]

DMA is mostly used for performance critical and data intensive communication, so mainly on the level of device data planes. Using the memory as intermediary however is not always the most efficient solution nor is asynchronous communication. [58–60]

2.1.2.3. Programmed I/O

When it comes to device control plane communication then synchronous and/or direct accesses are often preferred, for example for device setup, driver device feature negotiation, the DMA setup or configuration changes via specific device registers. Sometimes synchronous data plain communication is also desired and then so called programmed I/O is used. This is why devices are not only connected to the DMAC but also to the

CPU itself. Usually fast devices are plugged into slots on a dedicated PCIe (Peripheral Component Interconnect express) bus, that leads directly to the CPU. Slower devices are joined over the Platform Controller Hub (PCH) which is connected over the DMI (Direct Media Interface) bus to the CPU. Both busses more specifically lead to the I/O controller (IOC), the CPU's counterpart to the UMC for I/O device instead of memory accesses. In the following we will clarify how both those controllers actually know who a request is for. There are two mechanism that can be used, port mapped I/O (PMIO) or memory mapped I/O (MMIO). [58, 59, 84]

2.1.2.3.1. Port Mapped Input/Output The basic idea of port mapped I/O is that memory and devices are mapped into isolated address spaces and that the CPU has to use separate instructions to indicate which one is meant. [59, 91, 92] See figure 2.2 for an example.

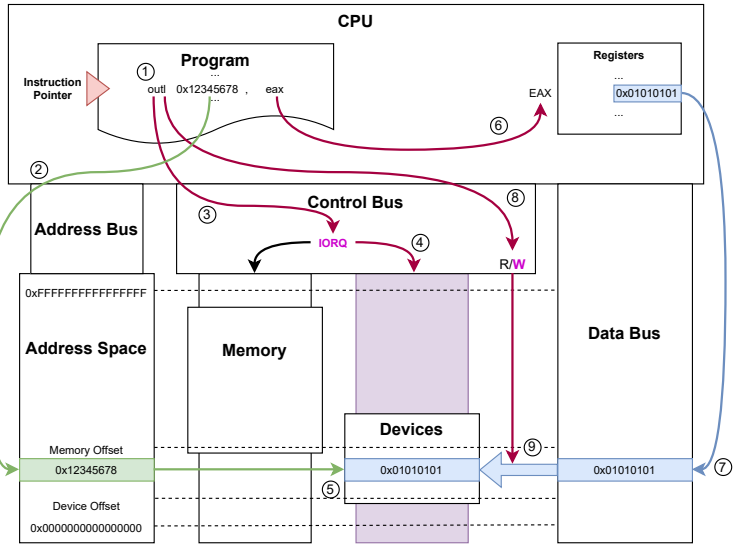


Figure 2.2.: Abstract depiction of Port Mapped Input/Output (PMIO). The program runs a special instruction for PMIO access ①. The specified address is put on the address bus ②. Because of the PMIO instruction ③ the I/O request bit (IORQ) is set for device access ④, hence the address references a value in the separate device address space ⑤. The second argument of the instruction is the register ⑥ those content should be put on the data bus ⑦. The `outl` furthermore commands the control bus to set the write (`W`) bit ⑧ which causes the value on the data but to be written to the specified address ⑨. [59, 91, 92]
Image inspired by [93].

Say the program wants to write 4 B of data from the `EAX` register to a device register mapped to address `0x12345678`. To accomplish this the program uses the instruction `outl 0x12345678, eax`. The `0x12345678` will be put on the address bus. Because of the mapping of devices and memory to their respective address spaces, it is already clear

that a device is addressed. If the address were a little higher however it would not. The `outl` however in contrast to a `mov` instruction for example will set the `IORQ` (I/O request) bit on the control bus selecting the device address space instead of the memory address space. It will put the data in `EAX`, so `0x01010101`, on the data bus and furthermore set the `W` (write) bit on the control bus. This will indicate to the IOC, that the data on the data bus should be written to the device. [59, 91, 92]

Note that some systems also implement PMIO with separate busses for memory and devices. Today PMIO is considered outdated and mostly around to handle legacy devices. It never received instructions beyond 32 bit during in AMD's x86_64 architecture specification, because it was being replaced by a simpler mechanism, memory mapped I/O. [59, 91, 92]

2.1.2.3.2. Memory Mapped Input/Output A 32 bit address space can address up to 4 GB. Since systems with that much memory already existed, having the devices mapped into a separate address space was unavoidable. But with the inception of 64 bit architectures the address space became so humongous that even today a computer with enough main memory to exhaust it is unimaginable. This opened up the opportunity to map devices into the same address space as memory and thus being able to use the very same instructions for accessing both. This technique is called memory mapped I/O. [58, 84, 94, 95] See figure 2.3 for an example.

Say the running program wants to read 8 B of data from a device register mapped to the address `0x0000FF0012345678` into the register `RDI`. Just as if it where from the main memory it uses the instruction `mov rdi, [0x0000FF0012345678]`. The address that is put on the address bus can only map to either the main memory or some device. In this case it is a device, so the IOC will answer it. The enabled read bit on the control bus will tell it, that the data is supposed to be read, so it just puts it on the data bus. From there the CPU fetches it into the designated register. [58, 84, 94, 95]

2.2. QEMU/KVM Devices

2.2.1. Context Switches

The operating system (OS) is the main program that is responsible for managing the systems resources and enabling the user to spawn and run multiple programs at the same time. To ensure its secure operation, its core runs isolated from the rest of the programs. This separation is reinforced in hardware, for example by the CPUs protections rings. While the operating system core or kernel runs in the most priviledged ring 0, user programs run in the least privileged ring 3. Anytime a user program wants to do something potentially dangerous that could interfere with the system's or other program's operation and thus needs to be overseen by the kernel, it will ask the kernel with a so-called system call (syscall) to do this. The program execution running in ring 3 will then be stopped and the system switches to ring 0, to let the kernel handle the

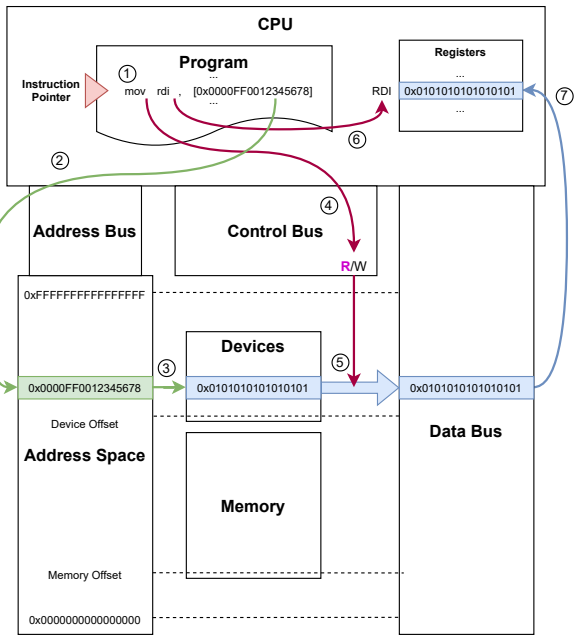


Figure 2.3.: Abstract depiction of Memory Mapped Input/Output (MMIO). The program executes a normal `mov` instruction ①, just as if it would access memory. The address to be read from is put on the address bus ② and can only point to a device or memory because their are mapped into the same address space when using MMIO. Here it points to a device register ③. Because the value should be read the read (`R`) bit is set on the control bus ④ fetching it on the data bus ⑤. The first argument of the `mov` instruction is the target register ⑥, so the value on the data bus is written into it ⑦. [58, 84, 94, 95]
Image inspired by [93].

request, execute the operation on behalf of the process, and finally return to the program execution. Switching between the so-called user space, where the user programs run, and kernel space, where the kernel runs, is called a context switch. [96–98]

2.2.2. Hardware-Assisted Virtualization

Just as user programs are seperated from each other and especially from the kernel they run on, it is imperative that virtual machines (guests) run isolated from each other, and most importantly cannot harm the host system or virtual machine monitor (VMM or hypervisor) that runs them. In the past ensuring that the guest doesn't overstep its boundaries was ensured by techniques like binary rewriting or again involving the protection rings, but these mechanisms have several drawbacks like performance penalties or lacking support for nested virtualization, to name a few. This is why modern CPUs assist host kernels with virtualization. Intel's virtualization extension VT-x added the so-called VMX root and non-root modes. Those can be thought of as a

second dimension in the protection ring structure. See the underlying table structure in figure 2.4 for reference. Ignore the details within, we will come to them later. [19, 24, 25]

The key of VMX is the so-called VM control structure (VMCS), which holds the state of a vCPU. It is used to store and restore said state when the execution context is switched between different guests or between host and guest. The host can manage these structs once it entered VMX operation and thus also VMX root mode with the `VMXON` instruction and not after it left it with the `VMXOFF` instruction. Once it has configured the VMCSs for a guest, it can start the guest with the `VMLAUNCH` instruction. The guest's vCPUs will then run isolated in VMX non-root mode. When a guest wants to give control back to the host it does a so-called VM exit which switches the execution context back to VMX root-mode. The VMM can then determine the reason for the exit and react. Once it is done, it continues the guest execution in VMX non-root mode with the `VMRESUME` instruction. This is also called VM entry. [99]

Note that AMD's virtualization extensions add similar mechanisms to their CPUs. [100]

2.2.3. Kernel-based Virtual Machine

Hypervisors commonly need components like a memory manager, a process scheduler, network and I/O stack, and many more to achieve their goal. If they want to make use of hardware-assisted virtualization they furthermore require access to priviledged instructions, like the VMX instruction set. All these requirements are already established in the Linux kernel. This thought is what lead to the development of the 2007 released Kernel-based Virtual Machine (KVM), an open source virtualization technology that turns the Linux kernel itself into a hypervisor. By default it is contained in the kernel module `kvm.ko`, which is installed out of the box into common Linux distributions. It is mainly build for CPU and memory virtualization on x86_64 systems and automatically harnesses Intel's or AMD's virtualization extensions, depending on which CPU the host kernel is running on, using the kernel modules `kvm-intel.ko` or `kvm-amd.ko`. It provides an extensive API through the virtual device `/dev/kvm` which can be accessed from user space via IOCTL (I/O control) calls. Apart from its respectable performance and free availability on every Linux machine, it has a rich feature set like SELinux- and sVirt-based security, live migration, resource control, real-time extensions, and many more. [101] This is the reason it is, as one of the mostly used hypervisors today, also part of some of the big cloud providers' architectures [29–35].

What KVM does not provide is virtualized hardware. This is where most often QEMU helps out [102].

2.2.4. QEMU

QEMU (Quick Emulator) is an open source machine emulator and most commonly used for system emulation in user space. While QEMU can emulate an entire system it is usually combined with KVM for running guests to achieve much better performance.

In this case QEMU provides the overarching machine model including CPU, memory, system bus and devices. It does however only emulate the latter two, while it lets KVM run the vCPUs and handle the guest's memory. [103]

2.2.5. Guest Device Access

We now combine the knowledge from the previous sections to understand how the device access works for a virtual machine. We first discuss guest DMA and then guest PIO in the form of guest MMIO.

2.2.5.1. Guest DMA

For guest DMA KVM provides shared memory regions that both the guest as well as QEMU thus the emulated device can access. The DMAC and IOMMU can be virtualized but also hardware assisted, especially when paravirtualized or passthrough devices are involved. We will see an example later on, when dissecting how VirtIO devices work. Just like physical devices most emulated or paravirtualized ones realize their data plane almost entirely using DMA. [20, 48, 49, 55–60]

2.2.5.2. Guest MMIO

For control plane access many virtual devices, like physical ones, combine both DMA and MMIO. The virtualization of MMIO however is a bit more complicated and also crucial to understand for the remainder of this work. We will therefore explain it in more detail with the help of the example shown in figure 2.4.

Assume we have a physical machine with an Intel VT-x enabled CPU running a Linux host system. QEMU runs in user space and uses KVM operating in kernel space to run a guest with hardware-assisted virtualization, meaning the host runs in VMX root mode and the guest in VMX non-root mode. An application in the guest's user space wants to access a device register emulated by QEMU. It will call the device driver running in its kernel space. The driver accesses the address to which the respective register has been mapped to via MMIO. Because this device is emulated the vCPU can't actually access the device directly and will therefore trigger a VM exit. The CPU saves the state of the vCPU in the corresponding VMCS and switches back to VMX root mode where the vCPU execution in the host kernel's KVM returns. KVM runs the return code against a `switch case` and finds that the VM exit was I/O related. Therefore it will cease its execution as well. This will cause a context switch back to the host user space where the IOCTL call, with which QEMU invoked the execution of the guest's vCPU, returns with the exit code `KVM_EXIT_MMIO`. Based on this QEMU picks out the device model to which the accesses' guest physical address belongs and calls its handler function to read or write the register. Once this is done QEMU's main loop invokes the vCPU execution again via KVM. KVM issues a `VMRESUME` instruction which causes the CPU to restore the

vCPUs state from the VMCS and switch back to VMX non-root mode completing the VM entry and giving back control to the guest. [48, 61–65, 104, 105]

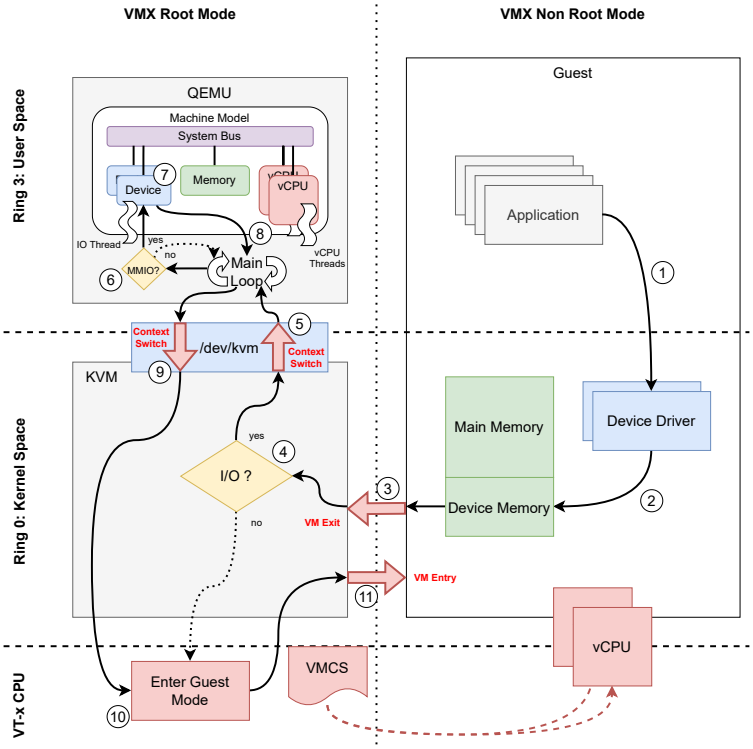


Figure 2.4.: Depiction of a PIO access of a KVM guest on an Intel VT-x CPU to a QEMU device. An application in the guest wants to access a device and thus communicates this to the driver in kernel space ①. The driver uses PIO to access the device memory ②. This causes a VM exit ③ and KVM checks the exit reason ④. Because it cannot handle the PIO request itself, it exits back to QEMU ⑤. QEMU then checks the exit code ⑥ and dispatches the request to the respective device ⑦. When handled QEMU returns to the main loop ⑧ and resumes KVM ⑨. KVM instructs the VT-x CPU to enter guest mode again ⑩. It will restore the vCPU state from the VMCS (VM control struct), and then do a VM entry ⑪. [48, 61–65, 104, 105]
Based on [104].

3. Analysis

While the last chapter lead up to the understanding of how device access is accomplished in QEMU/KVM guests, this chapter will start off with explaining how this influences the performance characteristics of devices specifically for MMIO and focus on latency as the key measure. We will discuss the existing optimizations, ioeventfd and irqfd, how they improve performance but also what restraints them. Finally we will introduce ioregionfd and explain how it overcomes those limitations.

3.1. Performance Impact of Guest Device Accesses

3.1.1. Efficiency of Guest DMA

As we have seen in the previous chapter, DMA is rather easy to implement for virtual machines. Besides the simple realization it profits greatly from its asynchronous architecture performance-wise, especially with our modern multicore CPUs. The guest doesn't expect to get data right away, thus the vCPU can keep running on one CPU core. Meanwhile, the virtualized device can work on the request on another core and once finished signal the vCPU with an interrupt in a lightweight manner. Ideally when there is not much load on the host there is no need for any CPU mode switches nor context switches during the parallel operation of the vCPU and the virtualized device. This is very different for guest MMIO accesses. [20, 48, 49, 55–60]

3.1.2. Expensive Operations During Guest MMIO Access

When we break down the guest MMIO access shown in figure 2.4 into a sequence diagram we roughly get figure 3.1.

Pay close attention to the crossings of the user and kernel space boundary as well as the VMX root and non-root boundary. From previous literature we know that Linux context switches alone easily cost several hundred of CPU cycles resulting in a delay of several microseconds at best. If the cache is under pressure this time can shoot up to a couple hundred microseconds. Similarly a CPU mode switch has been shown to cost several hundred CPU cycles as well, adding up to many thousands off cycles for handling an EPT violation related mode switch for example. Assuming the mode switch behaves similar under cache pressure and taking into account that we have two context and two mode switches per MMIO access, it is easy to see how the time for the entire operation can reach the order of milliseconds. [98, 106–113]

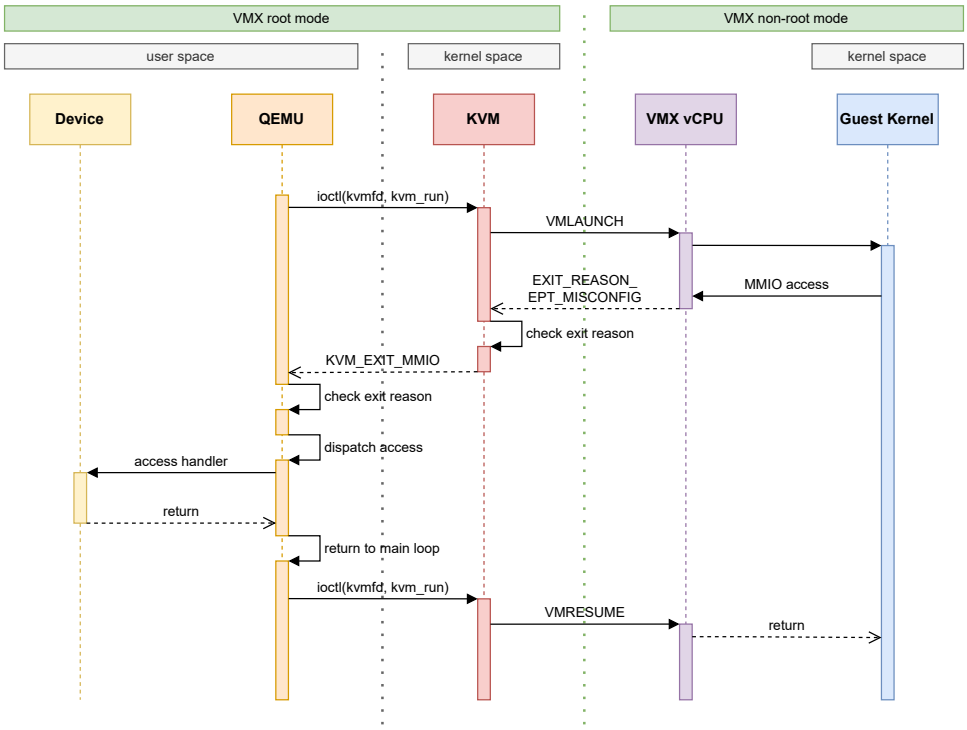


Figure 3.1.: Sequence diagram of the current MMIO handling of a QEMU/KVM guest. When the guest kernel accesses a device register via MMIO, the VMX vCPU exits back to KVM. KVM checks the exits reason and, because it is I/O related, exits back to QEMU. QEMU checks the exit code too, to find an MMIO request. It dispatches the request to the particular access handler and returns to the main loop, which invokes KVM again. KVM resumes the vCPU to give control back to the guest. [19, 48, 61–65, 99, 104, 105]

What remains in the sequence diagram, except the context and mode switches, are first the exit code checks and access dispatch in QEMU, both are realized with simple `if` and `switch` statements, thus should have negligible overhead, and second the access handler itself. How expensive the execution of the access handler is, depends very much on how the device emulation is implemented. Typically, this will be a read or write of a single device register, but even if it would involve more data it is safe to assume that also more data would pass the kernel user boundary. Since this has been shown to notably slow down the context switches due to increased cache pressure, the context switches should still remain the dominant part of the runtime of the MMIO access. Despite that is a slow device access handler not a problem of the virtualization, but would also happen if the emulated device would be used directly on the host, thus can be neglected as well. [63–65]

We conclude that the dominant operations in terms of runtime are likely the context switches and the CPU mode switches.

3.2. Optimizations for Guest MMIO

After getting an idea of the performance impact, we now turn to existing optimizations. We introduce eventfd and its offsprings ioeventfd and irqfd, and explain their benefit and limitations. We then explain what ioregionfd brings to the table to be a promising foundation for future optimizations.

3.2.1. eventfd

eventfd is a signaling mechanism between user and kernel space introduced into the Linux kernel in 2007. The affiliated system call creates an eventfd object containing an unsigned 64 bit integer counter and returns a file descriptor, referencing to it, to the caller. Reading from the descriptor returns the current counter, writing increments it by the supplied value. The user space application can use `poll`, `select` or similar syscalls on it, and close it once it no longer needs the eventfd. The kernel can also alter the counter. Based on these operations eventfd implements a lightweight omnidirectional notification mechanism. The kernel can notify the user space application and vice versa. The user space application can also wait for kernel notifications. Despite the initial eventfd system call this technique does not involve any context switches. [114, 115]

3.2.2. irqfd

Since KVM is responsible for CPU virtualization it also provides virtual interrupt injection modelled after native platforms such as x86. This can be used by emulated devices to inform the guest's vCPU about the completion of a DMA operation for example. Before eventfd this was handled in a synchronous manner by the kernel, meaning every interrupt from a QEMU device for example had to go through KVM adding significant overhead for both QEMU and KVM. This motivated the creation of irqfd in 2009, an eventfd-based mechanism to bind a user-space-accessible file descriptor to a specific entry in the guest's interrupt descriptor table (IDT). This allows device emulators like QEMU for example to signal the guest without syscalling to KVM. [11]

3.2.3. ioeventfd

The performance impact of guest MMIO, we have seen above, is obviously all the more problematic, when the access in question is nothing more than a notification to the device, like when initiating an out-of-band DMA request. These kind of accesses are usually also handled asynchronously on the native system, and this lead not long after irqfd to the development of ioeventfd. As the name suggests it is also based on eventfd, but uses it the other way around than irqfd. It allows a user space application like QEMU to attach an eventfd to a specific memory region, like a notification register of an emulated device, and hook up a handler function to it. This way write accesses to this region by the guest will enable KVM to asynchronously trigger the handler function without exiting back to user space. [12, 13]

3.2.4. ioregionfd

irqfd is designed for notifications to the guest. While this has merits for DMA it is unsuited for MMIO/PMIO accesses. ioeventfd goes the other direction and is used in a wide range of virtualized devices nowadays. It shows clear performance benefits for asynchronous MMIO/PMIO writes to notification registers for example, but that means there are also still many other MMIO/PMIO accesses that still go over the slow path involving the KVM exit. This urged kernel developers to propose a new more general mechanism called ioregionfd. It allows user space programs to bind file descriptors to memory regions by calling a specific IOCTL. With this IOCTL the user space program has to provide a `kvm_ioregion` struct shown in listing 3.1.

```
struct kvm_ioregion {
        __u64 guest_paddr; /* guest physical address */
        __u64 memory_size; /* bytes */
        __u64 user_data;
        __s32 rfd;
        __s32 wfd;
        __u32 flags;
        __u8 pad[28];
};
```

Listing 3.1: Definition of `kvm_ioregion` struct in the patched Linux kernel in `include/uapi/linux/kvm.h` used to register ioregionfds via the `kvm_set_ioregion` IOCTL. It contains the physical address of the guest's I/O memory region in `guest_paddr`, the size of the region in `memory_size`, user supplied data that is on access passed to the VMM in `user_data`, and the file descriptors for the read and write channels in `rfd` and `wfd`, a bit field for settings in `flags`, and padding in the end in `pad`. [14, 74, 76]

This struct specifies the memory region by start address and length similar to ioeventfd. ioregionfd is however not based on eventfd but defines a custom wire protocol running over the two file descriptors handed to KVM with the `kvm_ioregion` struct to support both read and write operations. This protocol consists of commands, KVM sends to the VMM to forward an MMIO/PMIO access, and responses from the VMM either to confirm or return data. The exact definition is shown in listing 3.2.

```
struct ioregionfd_cmd {
        __u32 info;
        __u32 padding;
        __u64 user_data;
        __u64 offset;
        __u64 data;
};

struct ioregionfd_resp {
        __u64 data;
        __u8 pad[24];
```

```
};
```

Listing 3.2: Definition of the wire protocol in the patched Linux kernel int `virt/kvm/ioregionfd.c`. The `ioregionfd_cmd` contains the `info` bit field with the access type, access size, and response enable bit, `padding`, the `user_data` supplied during setup, the `offset` of the access within the memory region, and the `data` to be written in case of a write access. The `ioregionfd_resp` contains just returned `data` in case of a read access, and padding in `pad`. [14, 74, 76]

Note that the `info` field in the command struct is a bit field containing the command (read or write), the size to be accessed, if a response is expected, and reserved bits or padding at the end. Both read and write operations are handled synchronously, but the user space program has the option to set a bit in the `flags` bit field during ioregionfd setup to indicate that writes should be posted, thus making it basically act like ioeventfd. [14, 74, 75, 79]

4. Design

Based on the brief introduction of ioregionfd the last chapter ended with, we will start off this one by understanding what makes the new mechanism so promising for optimizing guest MMIO. We then continue with a short analysis of the one currently existing implementation of ioregionfd in QEMU, before discussing what other virtualized devices might benefit from it in general. After that we take closer look at VirtIO, the device family we taylor our implementation to. In the close we will then explain how we approach our implementation in general and what requirements we want to fulfill.

4.1. ioregionfd-enhanced guest MMIO

We previously identified the CPU mode switch between VMX root and non-root mode, as well as the `KVM_EXIT_MMIO` related context switches as the main performance bottlenecks of guest MMIO. Obviously no mechanism within KVM can avoid the CPU mode switch, because this is how guest MMIO/PMIO accesses are implemented in the CPU. If we however take closer look at a sequence diagram of an ioregionfd-enhanced guest MMIO access, see figure 4.1, and compare this to the usual method we showed in figure 3.1, we can see that despite handing over control to the virtualized device in user space synchronously, there is no context switch involved.
Note also that file-descriptor-based access forwarding has the ability not only to attach a file descriptor to a memory region, but also attach the specific access handler for the particular region to the other end. This avoids all the access routing within the VMM that context-switch-based forwarding incurs. The absence of the context switch furthermore means that the access handler can run on a different CPU core than the calling KVM thread, allowing the KVM thread to keep being scheduled right where it is.

While all of this is also true for the more lightweight ioeventfd mechanism, ioregionfd gets its complexity from simply implementing more general operations. With synchronous read and write operations, as well as posted writes, it is a promising candidate to give all the remaining MMIO/PMIO operations on emulated and paravirtualized devices a fast path, just as ioeventfd did for notification registers.

4.2. Remote Device

As many of the big VMMs, QEMU is build in a monolithic architecture for performance reasons, running a single process with one thread operating each vCPU via KVM. With the large amount of features it offers it is however also very susceptible to attacks. To

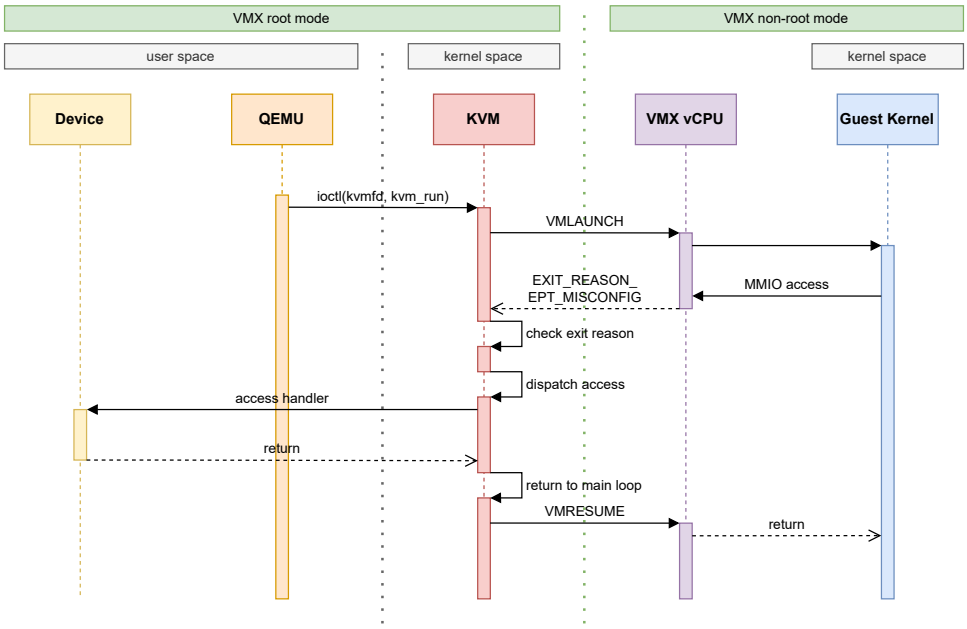


Figure 4.1.: Sequence diagram of the proposed MMIO handling of a QEMU/KVM guest. When the guest kernel accesses a device register via MMIO, the VMX vCPU exits back to KVM. KVM checks the exit reason and, because it is an access to a ioregionfd handled memory region, dispatches the access directly to the particular access handler. After that the KVM main loop resumes the vCPU to give control back to the guest. [14, 19, 48, 61–65, 74, 76, 99, 104, 105, 116]

mitigate this, QEMU now offers separating certain services from the main process. This general approach, called *multi-process QEMU*, makes it much harder for an attacker to exploit a vulnerability in a single service and take over the entire VMM, because the services are isolated into their own address spaces. A big part of QEMU's attack surface comes from its I/O devices, that due to their mere number can also include a lot of bugs. A very reasonable idea is therefore to split off part of the device emulation from the main process running the VMs. This is what QEMU's remote device does. It allows running a QEMU device in a separate process and attaching it to the main process, running the guest, via file descriptors handed to both processes via command line arguments. [77, 78]

As usual with security isolation techniques, this comes at the expense of performance, because now things like MMIO accesses get the additional overhead of IPC between QEMU's main and the remote device process. This motivated developing the first implementation of ioregionfd in QEMU for the remote device to lay the MMIO access path from KVM past the QEMU main process directly to the remote device process. Similar to the IPC communication, the file descriptors for ioregionfd are handed to both processes via command line arguments. At the time of writing this, the code comes with a few limitations, like only supporting a single ioregionfd per PCI device BAR (Base

Register Address) or a single file descriptor per device. The authors benchmarked their implementation on a storage device with the FIO (Flexible I/O Tester) [117] tool and found a small increase of IOPS (Input/Output Operations Per Second) for sequential read and write of about $1 - 2\%$, but a more significant one of $5 - 6\%$ for random reads and writes. Our guess is, that this difference comes from an increased amount of MMIO accesses during random I/O in comparison to sequential I/O. [79]

4.3. Potential Target Devices

When thinking about I/O devices for VMs or servers in general, both network and storage devices is what comes to mind first. In modern cloud infrastructures however we more and more see the use of lightweight sometimes even stateless virtual machines running a single application. The data they operate on often is stored on in external storage systems that are mounted over the network, while local storage devices are merely used for running the operating system. We would therefore argue, that improving the performance of virtualized network devices might have a slightly bigger impact on the overall performance of cloud applications. Therefore we want to focus on network devices in this work.

Despite that we can obviously only expect a measurable performance gain from applying ioregionfd to MMIO accesses that are executed regularly, ideally on every data plane interaction. For a network device this would mean registers that are read or written on every received or transmitted frame.

A last requirement, we see for an appropriate device to target, is that it is commonly used in virtualized environments. While any device might suffice to answer the academic question of whether ioregionfd is a viable solution, we want to make sure that our work is useful in practice. This is why we turned our attention to VirtIO.

4.4. VirtIO

VirtIO (Virtual I/O device) is a standard for para-virtualized guest I/O devices maintained by OASIS Open. It specifies a set of general bus drivers and device frontends for PCI, MMIO and channel I/O, as well as a variety of specific device backends and device drivers for common I/O devices like network adapters, block storage, consoles and more. This way no assumptions beyond support of the respective bus are made about the operating environment. VirtIO refrains from using any exotic mechanisms on the busses to make sure any device driver author can understand it and contribute. Its devices and drivers carry out a feature negotiation during setup to allow for forwards and backwards compatibility. Designed for efficiency and with the use of several optimization mechanisms, VirtIO is able to achieve remarkable performance, especially in comparison to traditional emulated devices. [48, 118]

For better insight into those points, take a look at the figure 4.2 depicting the inner workings of a VirtIO network device with the vhost protocol enabled.

QEMU runs a VirtIO network device backend on top of its VirtIO PCI frontend or proxy. Corresponding to that, the guest kernel runs the VirtIO network device driver on top of the VirtIO PCI bus driver. The proxy and bus driver establish device register accesses through KVM via PIO mostly for control plane operations. VirtIO's general data plane consists of a set of ring buffers, in case of a network device at least one for transmission and one for receival of packets. They hold descriptors for both available and used queue entries and a layout tailored towards cache locality. Those ring buffers are placed in a DMA region shared between guest and host. In case the vhost protocol is used, the host side of the VirtIO data path is offloaded to the host kernel, where it can directly forward network traffic from and to a TAP device for example. For efficient interrupts to the guest and notifications to the device, VirtIO uses both irqfd and ioeventfd. Note the modular structure, if a VirtIO PCI block device was considered instead, the QEMU device backend, the vhost backend and the guest device driver would change, as well as the device vhost is attached to, but the entire underlying bus setup would remain the same. [48–50, 118]

This combination of a straightforward standard, with an efficient and extensible design and last but not least good performance most likely is the reason why VirtIO became so popular and was also chosen as KVM's main platform for I/O a couple of years ago [119]. This obviously also makes it a modern and practical target for our work. A few tests with GDB and QEMU's tracing functionality on a VirtIO network device confirmed our assumption that MMIO accesses are rare during operation, when the PCI bus is used. When using VirtIO's custom MMIO bus however, we found frequent accesses during traffic, in fact the accesses appear to occur on every transmitted and received frame. Therefore it seems beneficial to concentrate our work on this bus type first. The VirtIO MMIO bus can be easily used with QEMU's MicroVM machine type, those purpose is mainly lightweight virtual machines (without PCI overhead), either in the context of embedded systems or resource critical cloud applications. While using VirtIO MMIO takes the burden of dealing with the more complex PCI bus away from us, it also means that we cannot easily bind DPDK to the guest devices, a caveat we will come back to later. [48, 120]

4.5. Our Approach

In the last section we learned that VirtIO devices have a frontend backend architecture, where only the frontend takes direct part in data and control plane communication. This also includes MMIO accesses from the guest driver. The mapped memory regions as well as the read and write handler functions are part of the bus specific, but device agnostic proxy, not the device itself. This means applying ioregionfd to those memory regions would implement this feature for all VirtIO devices that can and only when they are using the respective bus. In case of the VirtIO MMIO bus the frontend is the `VirtIOMMIOProxy`. In contrast to its more complicated PCI counterpart `VirtIOPCIProxy`, it contains just a single memory region for MMIO accessible registers, called `iomem`. This

is our target.

Aside from this main goal, there are a few other requirements we plan for our solution to fulfill if possible. First of all seems handing in file descriptors to QEMU as command line arguments a bit cumbersome for users and also unnecessary for our use case, therefore our implementation should create the file descriptors for ioregionfd internally. A second user related convenience would be to allow for a device specific activation of ioregionfd. And last but not least do we assume that MMIO setup is fairly similar for many QEMU devices even outside of the VirtIO family. If so we would like our code to be generic enough to be easily applicable to other devices as well. How we achieve all these goals will be the topic of the following chapter. There we will also take a closer look at the `iomem` region and what registers we actually apply ioregionfd to.

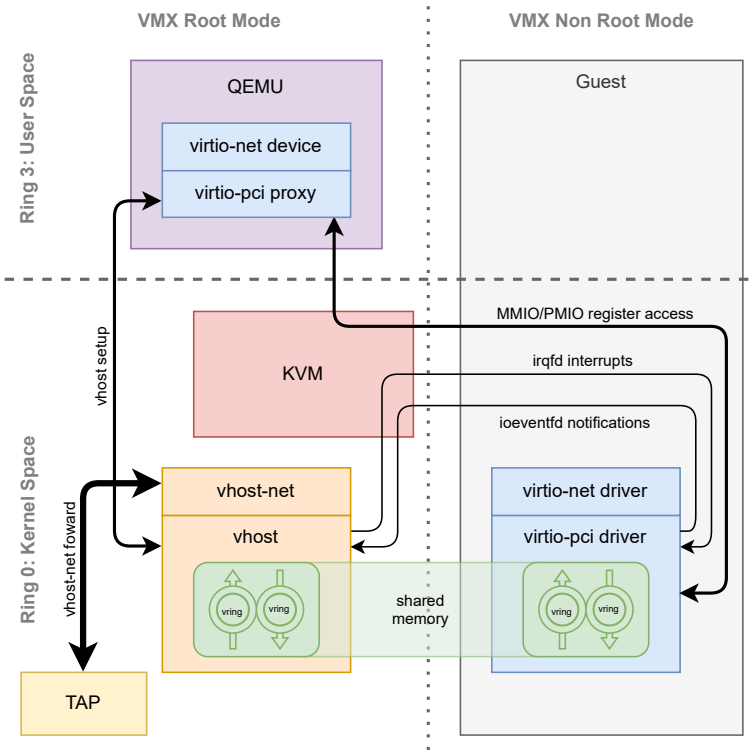


Figure 4.2.: Inner workings of a QEMU VirtIO PCI network device on a KVM host with an Intel VT-x processor and with the vhost protocol enabled. QEMU runs in host user space, para-virtualizing the VirtIO network device (backend) on top of the VirtIO PCI bus proxy (frontend). Corresponding to that, the guest kernel runs the virtio-net driver on top of the virtio-pci bus driver. Due to the vhost, protocol the VirtIO ring buffers, managed by the virtio-pci driver, are contained in a DMA memory region shared with the vhost kernel module in the host kernel. vhost-net rungs on top of this and forwards frames from and to a connected TAP device. Interrupts and notifications are handled by the means of irqfd and ioeventfd. PIO (MMIO/PMIO) accesses are caught by KVM and forwarded to QEMU. [49, 50, 99]

5. Implementation

In this chapter we will present important technical aspects of our implementation. We will start by where we hook into the VirtIO code, then describe in more detail how we establish the file-descriptor-based communication, show how we generalize the handler to fit more than just the VirtIO MMIO proxy, and finally address the registers of it that we handle via ioregionfd.

5.1. Initialization

Aware that implementing a first working version would take at least a couple of trials and errors, as well as debugging, we see it as imperative to be able to enable the usage of ioregionfd on a per device basis. This way our test guests would always have a working admin interface even if our code running for the test interface was still buggy. Therefore we add a boolean property to the `VirtIODevice` class, see listing 5.1.

```
static Property virtio_properties[] = {
    DEFINE_VIRTIO_COMMON_FEATURES(VirtIODevice, host_features),
    DEFINE_PROP_BOOL("use-started", VirtIODevice, use_started, true),
    DEFINE_PROP_BOOL("use-disabled-flag", VirtIODevice, use_disabled_flag, true),
    DEFINE_PROP_BOOL("x-disable-legacy-check", VirtIODevice,
                     disable_legacy_check, false),
    DEFINE_PROP_BOOL("use-ioregionfd", VirtIODevice, use_ioregionfd, false),
    DEFINE_PROP_END_OF_LIST(),
};
```

Listing 5.1: User-settable `VirtIODevice` properties in our custom QEMU built in `hw/virtio/virtio.c` with additional `use-ioregionfd` flag. It's boolean value is saved in `VirtIODevice.use_ioregionfd` and `false` by default. [121]

The property is `false` by default, but can be altered by the user via the QEMU command line by setting the `use-ioregionfd` option on the respective device, see the following example 5.2.

```
-netdev tap,vhost=on,id=admin1,ifname=tap1,script=no,downscript=no,queues=4
-device
    virtio-net-device,id=testif,netdev=admin1,mac=52:54:00:fa:00:60,mq=on,use-ioregionfd=yes
```

Listing 5.2: Example for a a QEMU command line for creating a vhost-enhanced multi-queue `virtio-net-device` on a specified TAP device, with specified MAC address, and with ioregionfd enabled.

The value is saved in the `use_ioregionfd` field of the `VirtIODevice` struct where it can be accessed from within QEMU.

The initialization of the `VirtIOMMIOProxy.iomem` memory region happens in the function
`virtio_mmio_realizefn` [122]. The obvious choice would therefore be to handle the ioregionfd registration similarly as it is done in the QEMU remote device [80], but unfortunately the `VirtIODevice` is not fully initialized at this point, meaning we cannot retrieve the value of the `use_ioregionfd` property.

Experiments with GDB reveal that at the point the `virtio_mmio_pre_plugged` is called, shortly before plugging the device to the guest, the `VirtIODevice` struct is fully initialized. This function also gets a `DeviceState` as argument, which gives it access to the `VirtioMMIOProxy` as well. Therefore we decide to place the ioregionfd initialization in this function.

The careful reader will likely notice that this means, we do not alter the memory region initialization in any way. In the QEMU remote device the usage of ioregionfd causes the regions memory operations to be set to `null`. In case we would register an ioregionfd for the entire region, a not so clean possibility would be to just null the operations afterwards in the `virtio_mmio_post_plugged` function. However as already mentioned in the last chapter and also addressed in more detail at the end of this chapter, we do not handle the entire memory region in one piece. This means the remaining registers will need these memory operations. Despite that is the nullification more of a cosmetic thing, them being in place also for registers that are handled via ioregionfd does not compromise the functionality. They cannot be invoked by the guest for those registers.

5.2. Socket

Just like the implementation in the QEMU remote device, we use the same file descriptor for read and write operations. This means we also have two file descriptors, one that is handed over to KVM with the `kvm_set_ioregion` IOCTL call, and one for the QEMU device. Similar to the remote device, we attach a `QIOChannel` (QEMU I/O channel) to the device's file descriptor and run all this in an `IOThread` (QEMU I/O thread). Due to the nature of the remote device however, the connected file descriptors are needed in two separate QEMU processes. Therefore the authors hand them to each process via command line arguments [79]. In our use case this is not an issue, meaning we can create the file descriptors in QEMU. Because the authors of the ioregionfd implementation in the remote device didn't provide any information on how they create the file descriptors, we follow the suggestions of the original ioregionfd proposal [14] and use a local socket pair, see listing 5.3.

```
    if (socketpair(AF_UNIX, SOCK_SEQPACKET | SOCK_CLOEXEC, 0, fds) < 0) {
        error_prepend(errp, "Could not create socketpair for ioregionfd");
        error_report_err(*errp);
        goto fatal;
    }
    ioregfd->kvmfd = fds[0];
    ioregfd->devfd = fds[1];
```

Listing 5.3: Socket pair creation for ioregionfd file descriptors in our custom QEMU built in `hw/virtio/ioregionfd.c`. `AF_UNIX` is used to create a local socket pair, with the options `SOCK_SEQPACKET`, to enforce consumers to read entire packets, and `SOCK_CLOEXEC`, to allow processes to avoid additional operations when settings this flag. Both file descriptors are saved in an `IORegionFD` struct. [121]

Note that we save both socket's file descriptors in an `IORegionFD` struct, which we will shed more light on in upcoming sections.

The communication happens between QEMU and KVM on the same machine, thus we use a `AF_UNIX` or local socket pair. With the `SOCK_SEQPACKET` option we enforce that consumers of the sockets read an entire packet on each read. Furthermore we set the `SOCK_CLOEXEC` to permit processes using the file descriptors to avoid additional operations to set this flag. Both options are necessary for the correct operation of the ioregionfd wire protocol.

5.3. Wire Protocol

As we explained above ioregionfd's wire protocol consists of commands send by KVM to the VMM to forward an access to I/O memory and responses send back confirming the transaction or returning requested data, see the struct definition shown before in listing 3.2.

```
struct ioregionfd_cmd {
        __u8 cmd : 4;
        __u8 size_exponent : 2;
        __u8 resp : 2;
    __u8 padding[7];
        __u64 user_data;
        __u64 offset;
        __u64 data;
} __attribute__((packed));

struct ioregionfd_resp {
        __u64 data;
        __u8 pad[24];
```

```
};
```

Listing 5.4: Definition of the wire protocol in our custom QEMU built in `include/hw/virtio/ioregionfd.h`. Compare this to listing 3.2 for the original definition. The difference is that here `ioregionfd_cmd` uses a GCC bit field to split up the `info` field in the original struct into its components, the access type in `cmd`, the access size as `size_exponent`, the response enable bit in `resp`, then `padding`, and then it continues like the original struct with the `user_data` supplied during setup, the `offset` of the access within the memory region, and the `data` to be written in case of a write access. The `ioregionfd_resp` matches the original definition and contains just returned `data` in case of a read access, and padding in `pad`. [121]

It is obviously of the essence to have the same protocol definitions in both KVM and the VMM, and this is not very difficult to realize for the struct overall, but beware of the `info` bit field. We were inspired by the remote device ioregionfd code to use GCC's (GNU Compiler Collection) bit field to allow direct access to the fields within and end up with what can be seen in listing 5.4. First however we had to find and fix inconsistencies between the KVM ioregionfd patch we use and the QEMU remote device patch we base our implementation on.

5.3.1. Debugging the Wire Protocol

Tracking down issues with the wire protocol can be quite challenging for various reasons, therefore we want to share our approach for the reader to use as guideline for debugging specifically ioregionfd's but also similar wire protocols.

Running incomplete VMM code sooner or later ends in the VM crashing or freezing up. In case of buggy device code it is not unlikely, that the guest does not even survive the boot process, making manual interaction with the device from the guest side impossible. Blacklisting the device or hot plugging it after booting might be an option. We make use of the fact, that we can choose for which register accesses our code is executed. We pin a single ioregionfd to a few bytes in the back of the `VirtIOMMIOProxy`'s `config` section, a part of the registers that we assume correctly to not be accessed by a `virtio-net-device` any time soon. This way we are able boot the guest and even use the network interface.

To invoke our code in a controlled manner, we use BusyBox's `devmem` tool, which allows for reading and writing to arbitrary device memory locations. The QEMU MicroVM architecture maps VirtIO MMIO devices to `VIRTIO_MMIO_BASE_ADDR` $+ 512 \cdot$ `device_index` with the base address being `0xfeb00000`. Reading from this address with `sudo busybox devmem 0xfeb00000` will return the first 4 B of the first VirtIO MMIO devices I/O memory, so the device's magic value. Note that the `devmem` command may require the kernel command line option `iomem=relaxed` to be set.

To find out where our ioregionfd enabled network device is mapped, we dump the detailed view of QEMU's device tree with the QEMU command line tool `info qtree`. The relevant parts of the NIC's entry can be seen in listing 5.5.

```
dev: virtio-mmio, id ""
  ...
  mmio 00000000feb02a00/0000000000000200
  bus: virtio-mmio-bus.21
    type virtio-mmio-bus
    dev: virtio-net-device, id "testif"
      ...
      use-ioregionfd = true
```

Listing 5.5: Relevant parts of the ioregionfd enabled `virtio-net-device`'s QEMU device tree entry. The device can be identified in the usually large dump by its device `id`, in this case `testif`, and then one can retrieve the device's bus index number, here `21`, and the address and size of the MMIO region, here `0xfeb02a00` and `0x200`.

The device entry can be identified by the `id "testif"` or by the `use-ioregionfd = true`. The `mmio` field shows that the I/O memory is mapped to the address `0xfeb02a00`. Note that the bus address matches the equation shown above with the 21 from the `bus: virtio-mmio-bus.21` as device ID.

With this information we can then access the ioregionfd handled part of the device memory, and observe the device side of the wire protocol communication by using GDB (GNU Debugger) on the QEMU process. Monitoring the other end of the "wire" is the actual challenge since it is part of the host kernel. While it is possible to debug a physical machine's Linux kernel with KDB (Kernel Debugger) over USB for example, we rather go with the to us well known approach of running the host as virtual machine and adding the QEMU argument `-s` allow for GDB to attach directly to the VM kernel. Aside from the complexities of nested virtualization, one thing to be aware of when debugging the kernel with GDB is, that it can only access the main kernel image's memory, not that of loadable kernel modules (LKM) which KVM is too by default. This means the host kernel needs to be compiled with the option `CONFIG_KVM=y` instead of the default `CONFIG_KVM=m`. We depict the general setup of this approach in figure 5.1.

When finally debugging a last thing to remember is, that the Linux kernel is highly optimized, meaning that control flow doesn't always run in the exact way it appears to be from the source code. Since the kernel does not compile without those optimizations a common workaround is to put in several intermediary values with `printk` calls to output them scattered over the source code about to be debugged. This forces the compiler to obey the given order. It is also an a workaround to make variables visible in the debugger, that would otherwise with optimized out.

5.4. Generic Handler

Similar to the remote device ioregionfd implementation, we define a struct called `IORegionFD`, see listing 5.6, to save context information surrounding a specific ioregionfd, either in setup functions or when routing an ioregionfd access. We save these structs in an array called `ioregfd` in the `VirtIOMMIOProxy` struct, see 5.7.

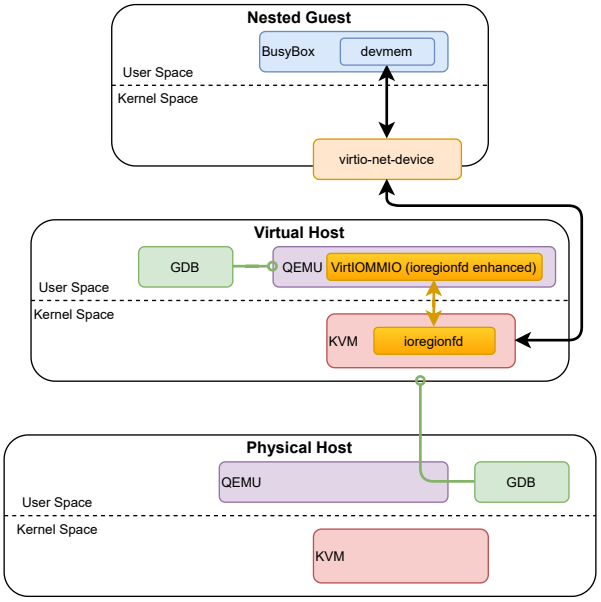


Figure 5.1.: Depiction of our ioregionfd wire protocol debugging setup. We use nested virtualization to debug the host kernel's KVM, one end of the wire protocol, with GDB. We also use GDB on the QEMU process for the nested guest to monitor the other end of the channel. We use the BusyBox `devmem` tool to trigger accesses to the ioregionfd-enabled memory region in a controlled manner.

```
typedef struct {
    int kvmfd;
    int devfd;
    gpointer opaque;
    uint64_t offset;
    uint32_t size;
    QIOChannel *ioc;
    AioContext *ctx;
} IORegionFD;
```

Listing 5.6: The `IORegionFD` struct definition in our custom QEMU built in `include/hw/virtio/ioregionfd.h`. It holds the file descriptors for the wire protocol in `kvmfd` and `devfd`, a generic pointer to the device frontend in `opaque`, `offset` and `size` of the handled region, as well as context information for the handler function in `ioc` and `ctx`. [121]

```
struct VirtIOMMIOProxy {
    /* Generic */
    SysBusDevice parent_obj;
    MemoryRegion iomem;
    qemu_irq irq;
    bool legacy;
    uint32_t flags;
    /* IoRegionFd */
    IORegionFD ioregfd[3];
```

Listing 5.7: `VirtIOMMIOProxy` device frontend struct definition in our custom QEMU built in `include/hw/virtio/virtio-mmio.h` containing three `IORegionFD` structs in the field `ioregfd`. [121]

A single variable would be possible as well, but an array is more general, since it also covers cases where multiple ioregionfds are used in the same device.

An example for the initialization of an ioregionfd with out code is shown in listing 5.8.

```
#ifdef CONFIG_IOREGIONFD
    if (vdev->use_ioregionfd) {
        ret = virtio_ioregionfd_init(&proxy->ioregfd[0],
                                     proxy,
                                     &proxy->iomem,
                                     0x0,
                                     0x50-0x0,
                                     virtio_mmio_ioregionfd_handler);
```

Listing 5.8: Initialization of an ioregionfd for a part of the `VirtIOMMIOProxy`'s I/O memory in our custom QEMU built in `hw/virtio/virtio-mmio.c`. First the general and device specific activation of ioregionfd is checked, then the first ioregionfd of the proxy is applied to the first `0x50` bytes starting at offset `0x0` of `iomem`. The generic `virtio_mmio_ioregionfd_handler` function is attached to the ioregionfd. [121]

First it is checked whether the ioregionfd is enabled in general and also for the particular device. If so an ioregionfd is initialized with the first `IORegionFD` struct in the proxy's `ioregfd` array. The handled region are the `0x50` bytes starting at `0x0` offset of the proxy's `iomem` memory region. The initialization is also provided with a pointer to the callback function that is registered as handler for the ioregionfd's QEMU I/O channel, over which the wire protocol commands come in.

However, instead of directly registering our core handler, we go a different path: When skimming the code for multiple QEMU devices, including the VirtIO MMIO and PCI devices, as well as the E1000 and RTL8139 emulated NICs, we notice that all their MMIO/PMIO read and write functions have the very same signatures, see listing 5.9.

```
static uint64_t virtio_mmio_read(void *opaque, hwaddr offset, unsigned size);
static void virtio_mmio_write(void *opaque, hwaddr offset, uint64_t value,
                              unsigned size);
```

Listing 5.9: Example of a common MMIO read/write function signature in QEMU. In this case of the VirtIO MMIO bus frontend. [122]

This gave us the idea to split the callback function into two parts:

1. The core handler gets the `IORegionFD` context struct and two function pointers with the signatures from above, retrieves and parses the incoming ioregionfd command, calls either the read or write function pointer to execute the command, and then sends back the ioregionfd response. This function is quite similar to but more generic than the one in the QEMU remote device.

2. The proxy callback only gets the `IORegionFD` context struct as `void` pointer to comply with the necessary signature for a QEMU I/O channel handler. It then calls the core handler with pointers to the original device or memory region

specific read and write functions. As an example, our VirtIO MMIO proxy callback is shown in listing 5.10.

```
static void virtio_mmio_ioregionfd_handler(void *opaque)
{
    IORegionFD *ioregfd = opaque;
    Error *local_error = NULL;

    virtio_ioregionfd_qio_channel_read(ioregfd,
                                       virtio_mmio_read,
                                       virtio_mmio_write,
                                       &local_error);
}
```

Listing 5.10: Example of a device specific proxy callback handler in our custom QEMU built in `hw/virtio/virtio-mmio.c`. It calls our device agnostic core handler with the correct read and write functions, in this case the previously shown `virtio_mmio_read` and `virtio_mmio_write` used for accesses to the `VirtIOMMIOProxy`'s I/O memory. [121]

This way only the proxy callback is device or region specific, while the core handler is completely device or region agnostic, allowing us to use the same core function for memory regions with different original MMIO/PMIO operations or even different devices.

5.5. Registers

Once the implementation is complete, our first approach is to register a single ioregionfd for the entire 512 B I/O memory of the `VirtIOMMIOProxy`. This does however cause the VM to crash immediately during boot. We therefore take a deeper look into the contents of the memory region. A summary of all registers contained is shown in table 5.1.

The column `Register` names the register, `Offset` is the offset of the register within the `iomem` memory region in bytes, and `Size` is the size of the register in bytes. `Access` describes whether the register is read-only (r), write-only (w), or read-write (rw). The last column `Description` contains a short description of the register's purpose.

In the beginning of the region we have basic device information like the device and vendor IDs, and after that registers for feature negotiation between device and driver. Then follow various registers for queue and shared memory management. In between we also have status and interrupt related registers. The second half of the 512 B region has no specified purpose, but can be used by the backend and driver for device specific registers.

Register	Offset	Size	Access	Description
magic value	0x0	0x4	r	Magic value, string "virt"
version	0x4	0x4	r	VirtIO device version
device id	0x8	0x4	r	VirtIO device ID
vendor id	0xc	0x4	r	VirtIO vendor ID
device features	0x10	0x4	r	Bit mask of device features
device features sel	0x14	0xc	w	Device feature set selector
driver features	0x20	0x4	r	Bit mask of activated driver features
driver features sel	0x24	0x4	w	Activated feature set selector
guest page size	0x28	0x8	w	Guest memory page size in bytes
queue sel	0x30	0x4	w	Queue selector
queue num max	0x34	0x4	r	Max. size of selected queue
queue num	0x38	0x4	w	Size of selected queue
queue align	0x3c	0x4	w	Used Ring alignment fo selected queue
queue pfn	0x40	0x4	rw	Guest page frame num. for selected queue
queue ready	0x44	0xc	rw	Ready bit for selected queue
queue notify	0x50	0x10	w	Queue notifier
interrupt status	0x60	0x4	r	Interrupt status
interrupt ack	0x64	0xc	w	Interrupt acknowledge
status	0x70	0x10	rw	Device status
queue desc low	0x80	0x4	w	Low and high 32 bit of selected -
queue desc high	0x84	0xc	w	queue's Descriptor Table address
queue avail low	0x90	0x4	w	Low and high 32 bit of selected -
queue avail high	0x94	0xc	w	queue's Available Ring address
queue used low	0xa0	0x4	w	Low and high 32 bit of selected -
queue used high	0xa4	0x8	w	queue's Used Ring address
shm sel	0xac	0x4	w	Shared memory region ID
shm len low	0xb0	0x4	w	Low and high 32 bit of -
shm len high	0xb4	0x4	w	length of shared memory region
shm base low	0xb8	0x4	w	Low and high 32 bit of -
shm base high	0xbc	0x4	w	shared memory region base address
config generation	0xfc	0x8	r	Configuration atomicity value
config	0x100	0x100	rw	Per-driver configuration space

Table 5.1.: Registers of the VirtIO MMIO I/O memory region. It starts with device information. Then follow registers for feature negotiation with the driver. After that registers for the queue management follow. In between there are the interrupt status and acknowledge registers. After the queue registers there are ones for shared memory management. Finally there is configuration space for device specific use. [48, 122]

To pinpoint the reason for the crashes during boot, we apply ioregionfd to each register separately. We find that for all but two registers the VM boots successfully:

1. Applying ioregionfd to the `queue notify` register causes the error:
   `kvm_mem_ioeventfd_add: error adding ioeventfd: file exists.`
   This means registering an ioregionfd on this register clashes with it already being handled by ioeventfd. This was to be expected.

2. Using ioregionfd on the `status` register leads to the error:
   `qemu-system-x86_64: ../qemu/softmmu/memory.c:1220:`
   `memory_region_transaction_commit:`
   `Assertion qemu_mutex_iothread_locked() failed.`
   So basically using ioregionfd here causes a mutex violation.

To ensure a combination of several registers is not problematic either, we then take advantage of our codes ability to handle multiple descriptors per device and in fact per memory region, and register ioregionfds for every part of the `VirtIOMMIOProxy`'s I/O memory but those two registers. This means the first ioregionfd handles bytes the `0x50` bytes starting at `0x0`, the second the `0x10` bytes starting at `0x60`, and the last one the `0x180` bytes starting at `0x80`. In this case the guest also boots successfully.

While we couldn't make estimates for every register at the time, it seems reasonable to assume that many of the remaining registers are not accessed on a regular basis, but more during device setup, the ones for features negotiation being an obvious example. We therefore use QEMU's tracing on the `VirtIOMMIOProxy`'s I/O memory read and write functions, so we start the guest with the QEMU command line options: `-trace virtio_mmio_read -trace virtio_mmio_write`. This reveals that only the `interrupt status` register seems to be accessed on a per-frame basis for a `virtio-net-device`, while most other registers are accessed during boot. For the sake of using ioregionfd most effective but minimally invasive we therefore decide to just apply it to the `interrupt status` register.

6. Evaluation

In this chapter we first explain what VirtIO device we use for evaluation, then give an overview of our measurement setup, before describing the concrete measurements we perform and their results.

From VirtIO's wide range of devices the ones that in our experience are used most often are the PCI block storage and PCI network adapter, more commonly known as `virtio-blk-pci` and `virtio-net-pci`. Their MMIO counterparts are the `virtio-blk-device` and `virtio-net-device` respectively. In line with our argumentation about the slightly bigger importance of virtualized network devices in contrast to local storage devices in modern cloud infrastructures, we decide to focus on the VirtIO network device.

6.1. Measurement Setup

6.1.1. Key Measure

The previously mentioned goal we aim towards with our approach is to reduce response times of regular MMIO/PMIO accesses to improve I/O performance. While this can ultimately result in a higher throughput, the more immediate consequence an thus key measure is the latency, so this is what our measurements focus on.

Latency is usually understood as the minimum time an operation takes. This is why when it comes to I/O devices, one is interested in the time it takes to process the request itself and not the involved data, since its transfer rate can already be estimated from the throughput of the device. Therefore latency is commonly measured by sending many very small requests, to max out the load from request processing and diminish the transfer's influence. In case of network interface cards this means that one sends many very small packets to isolate the time it takes from the sending up to the point where the first bit is being serialized on the wire, rather than the time it would take to serialize a full sized 1500 byte frame for example.

6.1.2. Packet Generator and Reflector

While I/O latencies are comparably large in comparison to response times for memory accesses for example, their are usually still in the millisecond and sub-millisecond range, thus low enough that software timers are too inaccurate for reliable measurements. Since commodity NICs however added hardware timestamping to their usual features software packet generators became competitive to the expensive hardware solutions.

Of those software generators we decide to go with MoonGen [123]. Besides the high precision latency measurements it can saturate a 10 Gb/s link with minimum frame size, offers LUA-scriptable packet processing, is open source, and is in our experience a reliable and easy to use solution. Like many other packet generators MoonGen draws its performance from DPDK (Data-Plane Development Kit) [124], a packet processing framework that directly binds to NICs DMA buffers from user space. [123]

The common setup for measuring NIC latency with a software like MoonGen consists of two machines, see figure 6.1: The load generator (often abbreviated LoadGen), that

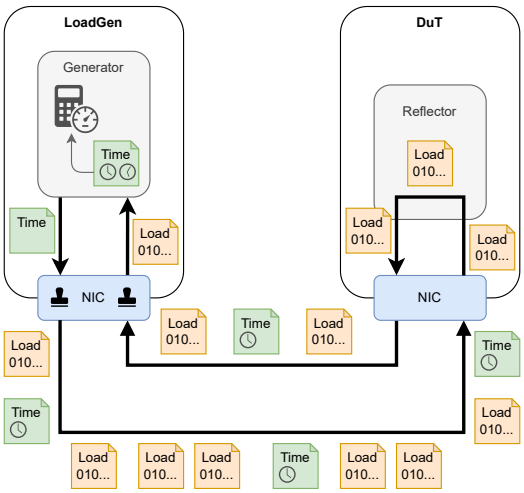


Figure 6.1.: Common setup of NIC latency measurements with a software packet generator like MoonGen. The load generator (LoadGen) machine runs the generator to send out many load and some timestamped packets on the wire. The device-under-test (DuT) machine uses a reflector to bounce every frame back to the LoadGen machine, where the measurement frames get timestamped again. The generator the calculates the round trip time (RTT) from the difference of the departure and arrival timestamps and approximates the latency with this value. [66, 123]

runs the packet generator to send out many frames, some of which timestamped on departure, to the second machine with the network card being measured. It is often referred to as device under test (abbreviated DuT). The DuT then reflects those load and time frames back to the best of its ability. The load generator then again timestamps the time frames on arrival and calculates the round trip time (RTT) from the difference of the two timestamps to estimate the latency. [66, 123]

For the reflector one could usually use MoonGen as well. DPDK's device binding however is limited to PCI devices, hence does not work for a `virtio-net-device`. We therefore wrote a tiny reflector program called `xdp-reflector` [125]. It is based on XDP (eXpress Data Path) [126], a high performance data path for bare metal packet processing, that works directly on the memory pages containing the received packets outside the drivers receive queue, and thus even before any other kernel-side processing occurs on the packet. An XDP program can be loaded and attached to a NIC with the

`iproute2` toolkit, and does not require binding a special driver to the NIC as DPDK does. Therefore it works just the same for non-PCI devices like the `virtio-net-device`. [124, 126, 127]

6.1.3. Test Bench

Benchmarking virtualized NICs on its own already adds considerable complexity to the previously shown test setup. Despite that we also plan to compare several different several virtualized NICs with different settings against each other, but also against the host interfaces. Therefore we want to take a moment to describe our test bench in more detail, see figure 6.2.

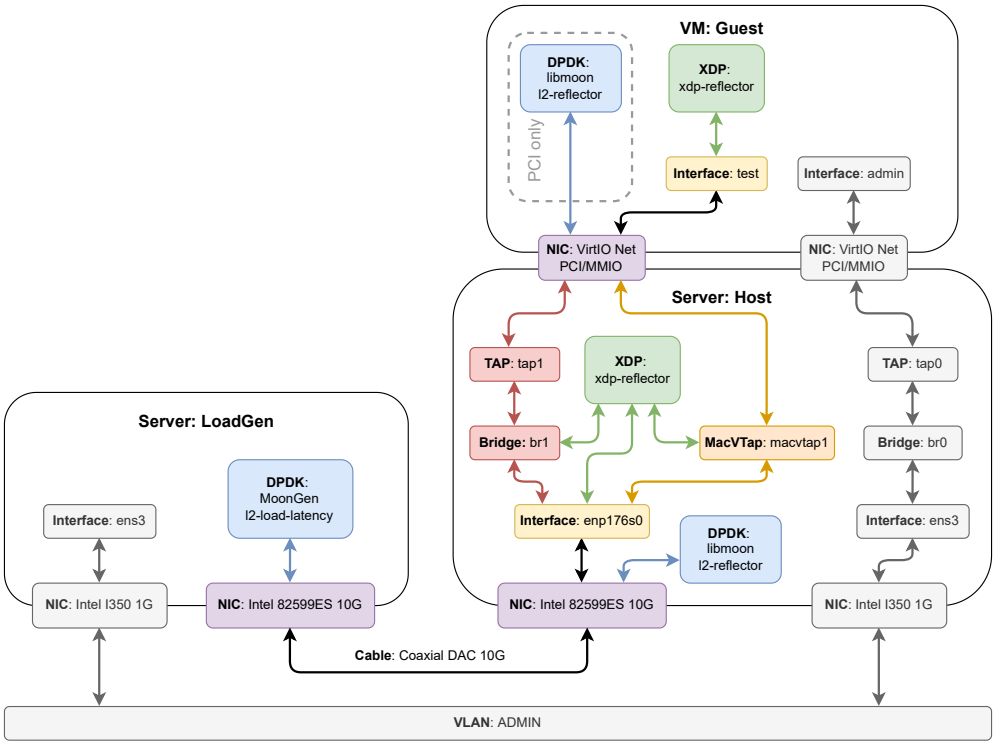


Figure 6.2.: Block diagram of our test system. We have the LoadGen machine running MoonGen for load generation on a 10G Intel NIC connected via DAC directly to the DuT machine. On the DuT guest we have several different virtual NIC setups, either via Linux bridge or MacVTap, also with VirtIO PCI or MMIO. On the guest we can reflect either with DPDK or XDP. On the DuT we can also run reflectors for comparison. Note that we only run one particular NIC and reflector setup at a time. All machines including the guest are also connected to our ADMIN VLAN for management purposes.

Our test system comprises two identical FUJITSU Primergy CX250 S2 nodes with each two Intel Xeon E5-2670 v2 CPUs running at up to 2.5 GHz, each two Samsung

M393B1K70CH0-CH9 8 GB 1333 MHz ECC DDR3 DIMMs, and each a Samsung 870 EVO SATA SSD as root disk. Both servers run on Debian 11 bullseye but while the loadgen server runs the stock kernel, the host system uses a custom kernel with the ioregionfd feature [128]. Both servers are connected to out administration VLAN via their on-board Intel I350 1 Gb/s NICs. Despite those both systems are equipped with an Intel 82599ES based 10 Gb/s NIC for the actual measurements. Both cards are connected with a 10 Gb/s coaxial direct attach cable (DAC) and therefore isolated from the rest of the network to avoid any disturbances.

On the loadgen server we bind with DPDK directly to the Intel 82599ES NIC and for measurements run a customized version of MoonGen's example `l2-load-latency.lua`. For measurements on the physical NIC we either attach DPDK and run the libmoon example `l2-reflector.lua` on the host, or we attach `xdp-reflector` to the corresponding interface. The host has our custom version of QEMU [129] installed to run the guest. The guest also runs Debian 11 bullseye and for the test presented below had 4 vCPUs and 4 GiB of RAM assigned. For SSH connectivity we connect the guest's admin interface to the admin VLAN over a Linux bridge. For the test interface we evaluate two different host network setups, first a bridge as for the admin interface and second a MacVTap both connected to the 10G NIC. For comparison we may also use the `xdp-reflector` on those interfaces to see how much latency they alone add. The guest's NICs are both `virtio-net-devices` or again for comparison `virtio-net-pci` devices. In case of PCI we can then also use the libmoon reflector instead of our XDP-based solution.

To manage this complicated setup and run all test cases automatically based of a configuration file, see A.1, we use a program specifically developed by us for this project called [130]. Note that it still contains a lot of hardcoded values and commands and is therefore not yet suitable for use outside of our one test system, at least not without some changes.

6.2. Measurements

In this section we now finally come to the actual measurements and results. We start with the raw performance of the physical NIC and use it as a baseline to judge the performance of the XDP reflector. We then continue with the measurements on the VirtIO devices. We take into account several different factors: the bus type, the host network setup, the use of the vhost protocol, and where possible the use of ioregionfd, as well as the available reflectors. In the end we look at three more devices, QEMU's e1000 and rtl8139 emulated NICs, as well as the VirtIO block device.

Note, that all MoonGen latency measurements, from which we also derive the packet loss, ran for 30 s and were repeated three times. All resulting plots show the average of those three runs. Despite that some measurements turn out to be very noisy, and due to the large amount of time it takes to retake all the measurements even for just a single plot, we were not able to get all of them to be clear enough to easily derive conclusions. Our final remark is, that most of the measurements appear multiple times in context of different variables we are looking at. Due to the large number of variables we consider, putting them all into one plot to avoid the redundancy would negatively affect the readability. An alternative would be to read off significant values from the plots, like the maximum load and final latency, but at the expense of loosing any other characteristics of the latency curve. Therefore we decide to keep the plots as they are.

6.2.1. Physical NIC

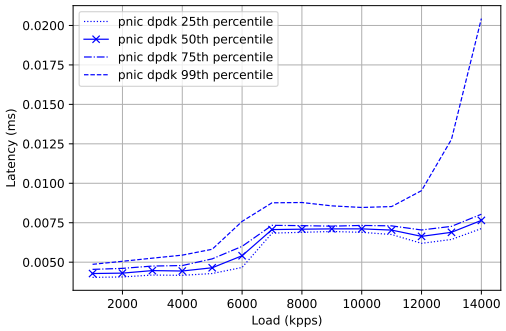
6.2.1.1. Raw Performance

With MoonGen's ability to saturate a 10G link with minimal frame size, libmoon's L2 reflector example (`reflector.lua`) script proves to be sufficient to properly measure the raw performance of the physical NIC. The resulting latency for different loads is shown in figure 6.3a.

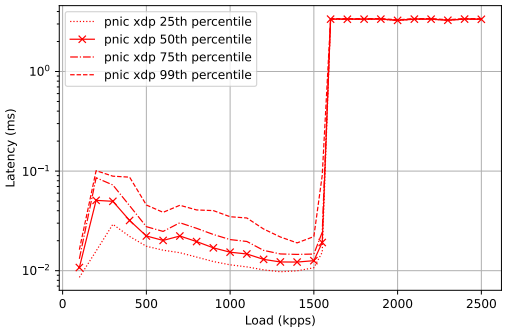
We can see that the latency at under 5 Mpps is very low, under 5 µs. Between 5 Mpps and 6 Mpps it then increases to about 7.5 µs and on average stays there until 14 Mpps. The reason is probably that the NIC's packet queues get longer, hence the frames have to wait longer to be processed. The only significant change towards the end is the jump of the 99th percentile to 20 µs. This is to be expected since 14 Mpps is close the maximum rate a 10G link can handle with 64 B frames, so the NIC approaches its limits. However, taking into account that the 75th percentile is still very close to the average slightly above 7.5 µs and the packet loss, which can be seen in figure 6.3c, staying at zero the entire time, the card is clearly capable of what it is supposed to.

6.2.1.2. XDP reflector

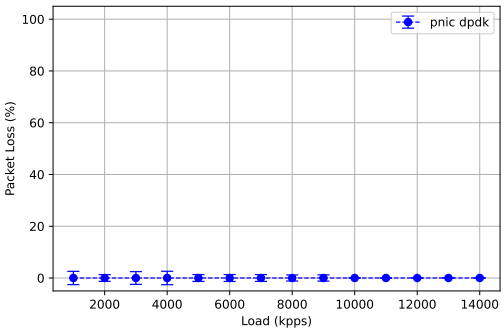
After the baseline being set by the raw performance, we now look at the same measurement but with the XDP reflector. This should give us a good idea of the overhead it causes. The latency for different loads is shown in figure 6.3b. Pay attention to the



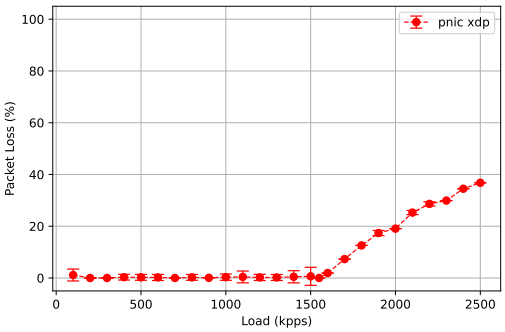
(a) Latency with libmoon reflector.



(b) Latency with XDP reflector.



(c) Packet loss with libmoon reflector.



(d) Packet loss with XDP reflector.

Figure 6.3.: Latency (top) and packet loss (bottom) of the physical NIC for different packet rates, with the DPDK-based libmoon reflector (left) and the XDP reflector (right). Frame size is 64 B.

logarithmic scale on the latency axis. The latency starts at about 10 µs at 100 kpps, which is already twice as high as with the libmoon reflector. It increases to 50 µs at 200 kpps and 300 kpps, and after that decreases slowly down to about 15 µs at 1.5 Mpps. The distribution is also broader that for the libmoon reflector, with the 99th percentile sometimes being twice as high as the average. At 1.6 Mpps the latency than jumps up significantly to about 2.5 ms and stays there for the rest of the measurement. We have found the maximum load the setup can handle. The packet loss, shown in figure 6.3d, confirms this. It starts at zero up to 1.6 Mpps and then increases in the expected geometric conversion towards 100%, one can expect from a network interface in an overload scenario.

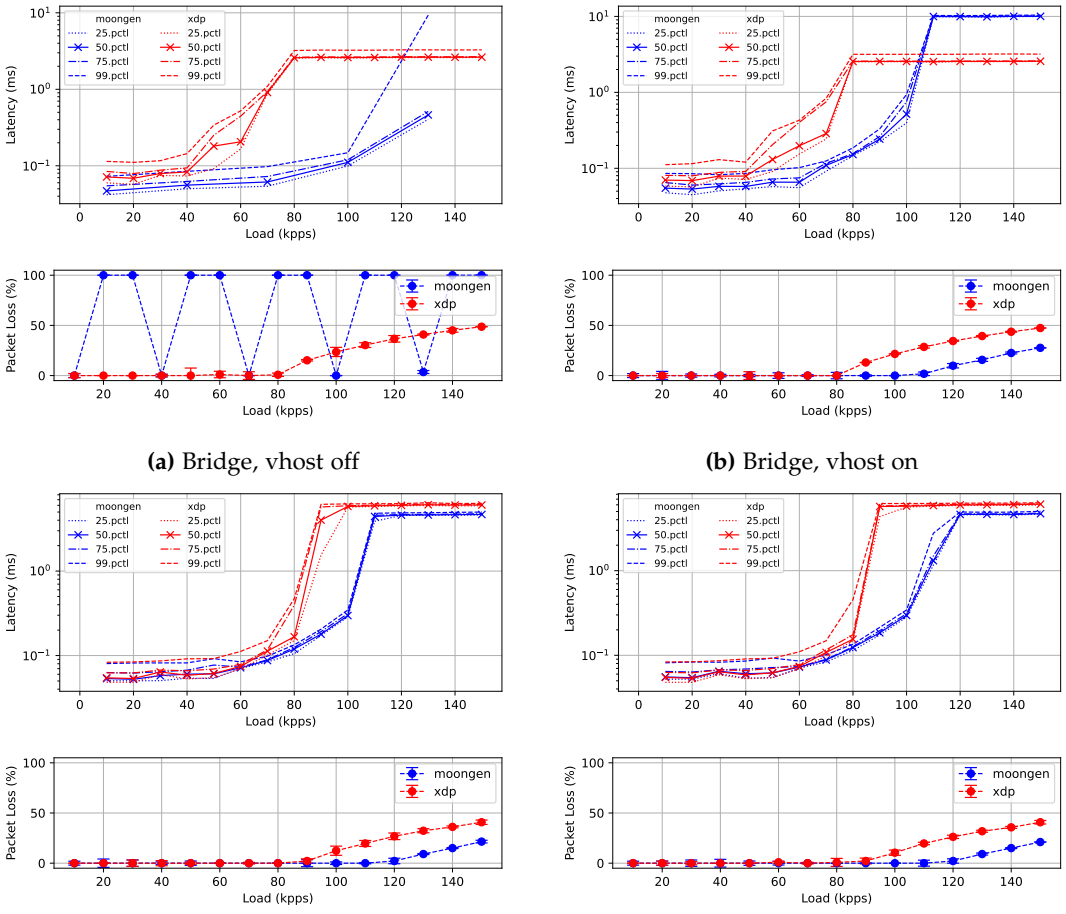
While XDP is apparently capable of much higher packet rates [131–133], our setup is not. We assume that our rather old CPUs are the main reason. We also don't fully exclude the network driver as possible cause. Even if this is a clear limitation in comparison to DPDK-based reflectors, it should be sufficient for the even lower packet rates we can expect from the usual emulated or para-virtualized NICs, we focus on in the following.

6.2.2. VirtIO Network Device

In this subsection we now thoroughly analyze the performance of the VirtIO network device with focus on the influence of ioregionfd. To properly understand those results in particular, however, we first look at various other parameters.

6.2.2.1. Reflector

First we look again at the reflector performance. Since we can only use DPDK on PCI devices, we compare both reflectors on the virtio-net-pci device. The results for both interface types and vhost settings are shown in figure 6.4.



(a) Bridge, vhost off

(b) Bridge, vhost on

(c) MacVTap, vhost off

(d) MacVTap, vhost on

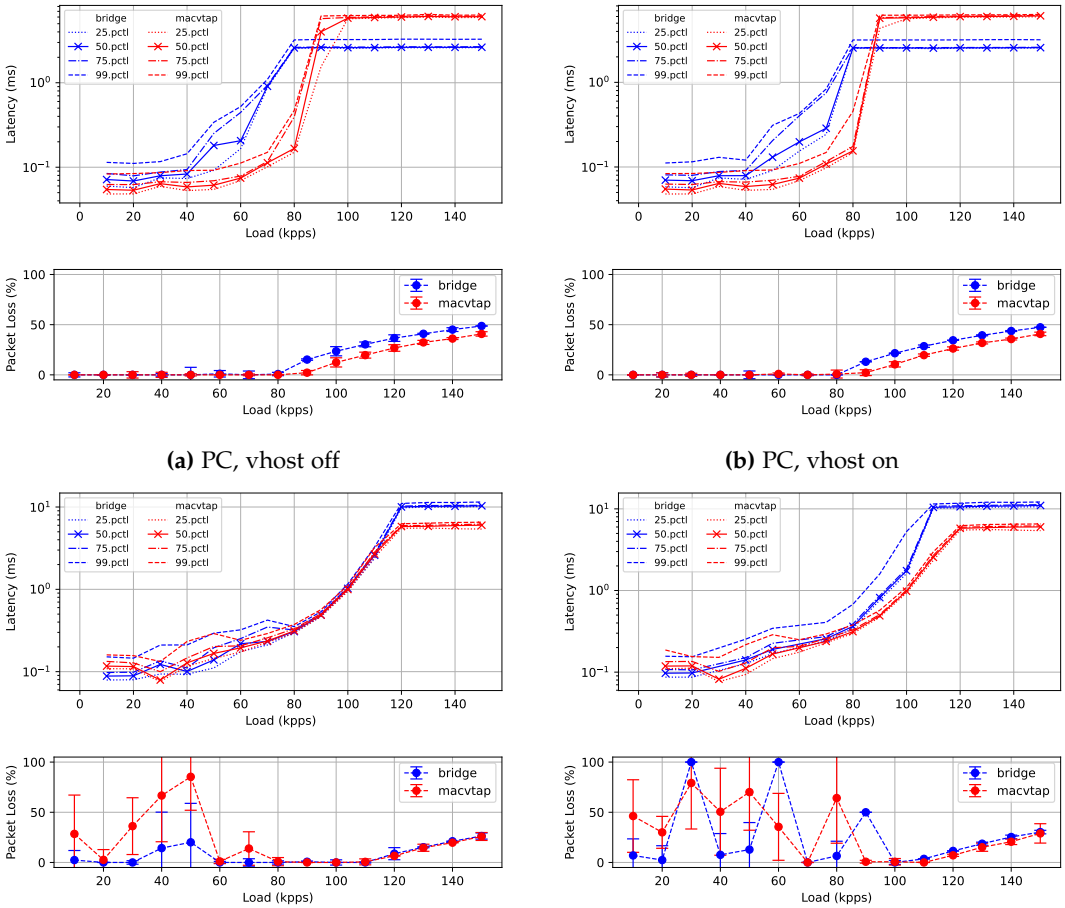
Figure 6.4.: Load latency and packet loss comparison for different reflectors (MoonGen vs. XDP), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, bus type is PCI, and ioregionfd is off.

We have the latency and packet loss plots for the Linux bridge on top and MacVTap on the bottom, vhost off on the left and vhost on on the right. A first observation is

the significantly lower performance, we expect in comparison to the physical NIC, even with the XDP reflector, which according to our previous argument should justify using it for guest measurements. Note that for the bridge with vhost off in plot 6.4a we didn't get a clean measurement. What we can gather from the rest of the plots however is, that the with the DPDK-based reflector the interface is capable of about 120 kpps to 130 kpps, while we only get around 80 kpps to 90 kpps with the XDP reflector. While for the MacVTap the latency is also lower for the libmoon reflector, with a final value of roughly 5 ms compared to about 6 ms for the XDP reflector, the final latency of for the bridge is counterintuitively higher for the DPDK-based reflector at about 10 ms compared to only 3 ms for the XDP reflector. We are aware, that XDP is said to sometimes cause packet loss [134, 135] at least more than DPDK, but we don't see any issues here. We believe that while the reflector certainly has an influence, the difference might be more to blame on the different drivers used in both cases. What we also see is a clear difference between the two interfaces, which we will cover now.

6.2.2.2. Bridge and MacVTap

Figure 6.5 compares the load latency and packet loss for the bridge and the MacVTap, with both bus types, PCI (on top) and MMIO/MicroVM on (on bottom), and vhost settings, vhost off (left) and vhost on (right).



(a) PC, vhost off

(b) PC, vhost on

(c) MicroVM, vhost off

(d) MicroVM, vhost on

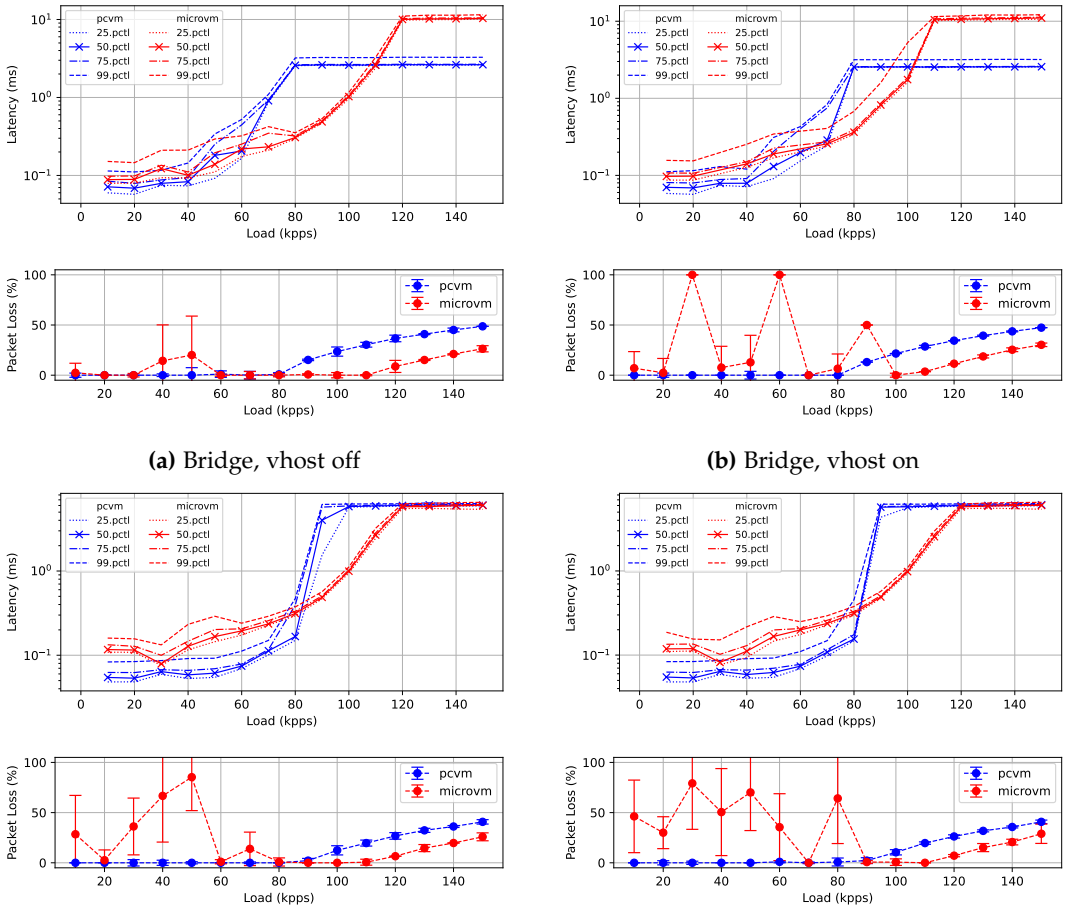
Figure 6.5.: Load latency and packet loss comparison for different interfaces (Bridge vs. MacVTap), PC (top) and MicroVM (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, reflector is XDP, and ioregionfd is off.

We expect the MacVTap to perform better, since it connects the guest interface directly to the physical NIC and circumvents the host kernel network stack in contrast to the bridge which sits in the host kernel. In case of the MicroVM, this is obviously the case. While both interfaces start with a similar latency behavior, the bridge strives towards a higher final latency, with 10 ms compared to only 6 ms for the MacVTap. Both interfaces reach their packet rate limit at roughly 110 kpps and 120 kpps, too. For the PCI-backed guest on the other hand the benefit of the MacVTap is more apparent before

the interfaces reach their load maximum. While the bridge finds it at 80 kpps and about 2 ms, the MacVTap is still close to 100 µs latency. 10 kpps later it however shoots up to 6 ms. Additionally taking into account the packet loss, there seems to be a clear influence of the bus type of the guest, so this is the angle from which we look at the data next.

6.2.2.3. Bus Type

Figure 6.6 shows the load latency and packet loss comparison for the two different system bus types, PCI and VirtIO-bus-based MicroVM, for both interfaces, bridge (on top) and MacVTap (on bottom), and vhost settings, vhost off (left) and vhost on (right).



(a) Bridge, vhost off

(b) Bridge, vhost on

(c) MacVTap, vhost off

(d) MacVTap, vhost on

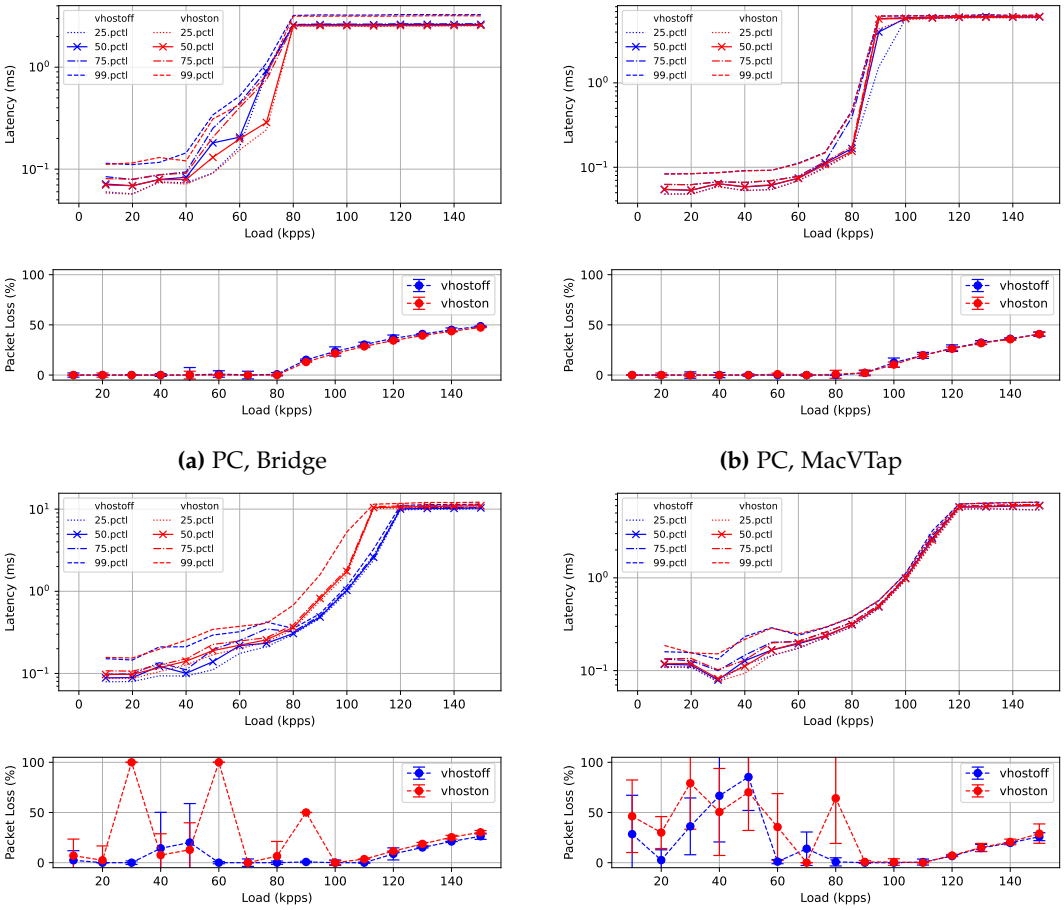
Figure 6.6.: Load latency and packet loss comparison for different machine types (pc vs. microvm), and therefore different bus types (PCI vs. MMIO), thus between virtio-net-pci (blue) and virtio-net-device (red), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, XDP is used as reflector, and ioregionfd is off.

We see that, except from the higher latency towards the maximum load for the bridge with 10 ms final latency compared to 6 ms for the MacVTap, the MicroVM performs very similar in all cases and hits the same load maximum of 120 kpps in three of four cases. The PCI-backed guest cannot keep up with this, and only reaches 80 kpps with the bridge and about 90 kpps with the MacVTap. Its final latency is the same for the MacVTap, but again surprisingly lower for the bridge at only 2 ms. Another interesting observation while the virtio-pci-device only drops packets once it reaches its load maximum, the virtio-net-device (MicroVM) displays significant packet loss before reaching its maximum without any pattern. After the maximum however, there are no signs of packet loss additional to the expected "overload" drop. Also when there is packet loss below the load maximum, than with a huge standard deviation. This is a very strange behavior, so we look closer at the MoonGen outputs and discovered, that the RX rate jumped apparently randomly between mostly between no and full packet loss, see A.2. We discover this very late because our measurements are fully automated. In the beginning our scripts parsed only MoonGen's histograms and we didn't even safe the outputs. Only after we noticed performance issues did we script parsing MoonGen's outputs to calculate the packet loss. The zero packet loss with the PCI-bus and the previous comparisons in 6.2.2.1 at least lets us rule out XDP as the cause to some degree. Note also, that we use a Linux version 5.19 for all the tests with the XDP reflector, which contains the previously mentioned fix for spurious XDP packet drops [135]. Our issues are also not an occurrence of loss just in the beginning of the measurement, which has been mentioned in the literature before [136], and doing initial warm-up measurements also did not fix this. While we don't know the root cause for sure, this comparison strongly suggests problems with VirtIO's MMIO bus or the virtio-net-device or driver. A possible explanation might be, that it configures the VirtIO rings differently than the virtio-net-pci device, which can impact the performance [137]. The large standard deviation of the packet loss may suggest that jitter is also involved here, which in our experience is quite a common occurrence when doing performance measurements on virtual machines [138]. We unfortunately did not have the time to investigate this further.

6.2.2.4. Vhost Protocol

A parameter we have only introduced on a theoretical level yet, when explaining the inner workings of VirtIO, is the vhost protocol. Figure 6.7 compares the load latency and packet loss behavior for vhost enabled and disabled, with PCI-backed guest (on top) and MicroVM (on bottom), bridge (left) and MacVTap (right).

We know that vhost moves the data plane of the virtual NIC to the kernel, where it can bind directly to the connected TAP device, so we would expect a gain in performance. In the measurements, we however don't see this. In three of four cases the load latencies look almost identical, while the packet losses behave at least similarly. The only exception is the MicroVM with the bridge, where we have a few cases of high packet loss for some packet rates, and a bit lower maximum load with vhost enabled. The packet loss also



(a) PC, Bridge

(b) PC, MacVTap

(c) MicroVM, Bridge

(d) MicroVM, MacVTap

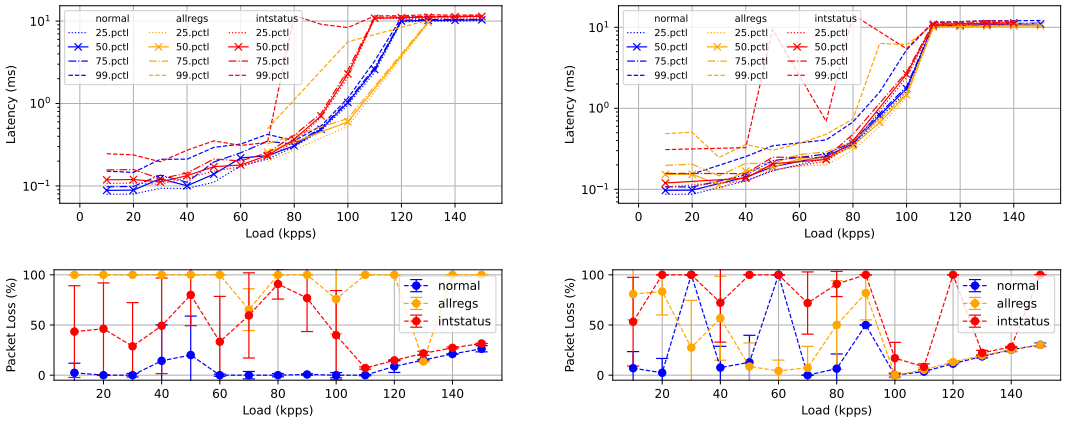
Figure 6.7.: Load latency and packet loss comparison for different vhost settings, PC (top) and MicroVM (bottom), Bridge (left) and MacVTap (right). Packet size is 64 B, reflector is XDP, and ioregionfd is off.

appears in the MacVTap case, which strongly suggests, that this is again to blame on the VirtIO MMIO bus. The difference in in maximum load is just 10 kpps, which happens to be the increment we used for our measurements. This could suggests, that the actual maximum load probably lies somewhere in between those two packet rate steps, and the value for the vhost enabled is lower merely by chance.

6.2.2.5. ioregionfd

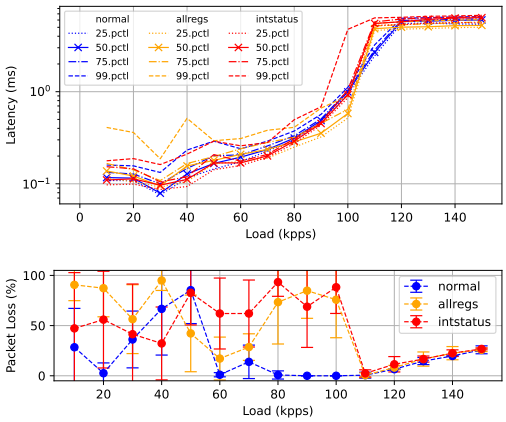
We now come to the parameter, we are more interested in, ioregionfd. Figure 6.8 compares the load latency and packet loss behavior for three cases: ioregionfd disabled, ioregionfd applied to just the `interrupt status` register, and ioregionfd applied to all registers that work, so all but `queue notify` and `interrupt acknowledge`. The figure

has the bridge plots on top and the MacVTap plots on the bottom, vhost is disabled on the left and enabled on the right.

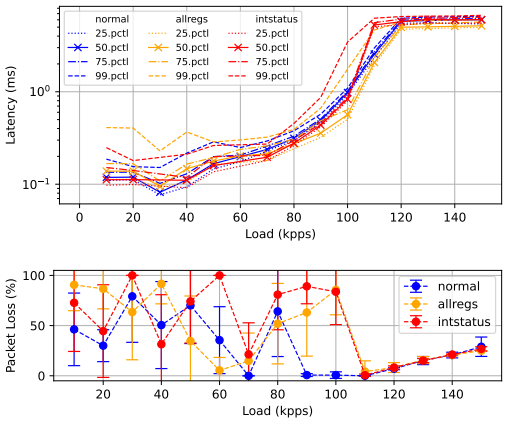


(a) Bridge, vhost off

(b) Bridge, vhost off



(c) Bridge, vhost off



(d) Bridge, vhost off

Figure 6.8.: Load latency and packet loss comparison for ioregionfd on and off, Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, bus type is MMIO, and reflector is XDP.

Since previous comparisons show, that neither vhost nor the interface type seems to have an effect on the final latency, and we also don't see many differences here, which are most likely to blame on packet loss and measurement inaccuracies, we focus on what is common to all plots. We see, that on average all implementations perform very similar. While just applying ioregionfd to the `interrupt status` register seems to have no or even detrimental effect, applying it to all registers appears to show minor improvements. When looking at the 99th percentile however, we see much higher spikes when ioregionfd is enabled. This seems to be consistent with the packet loss behavior, where we see a general trend towards higher drops with ioregionfd enabled. While we can certainly blame some of the packet loss on the circumstance, that we are using

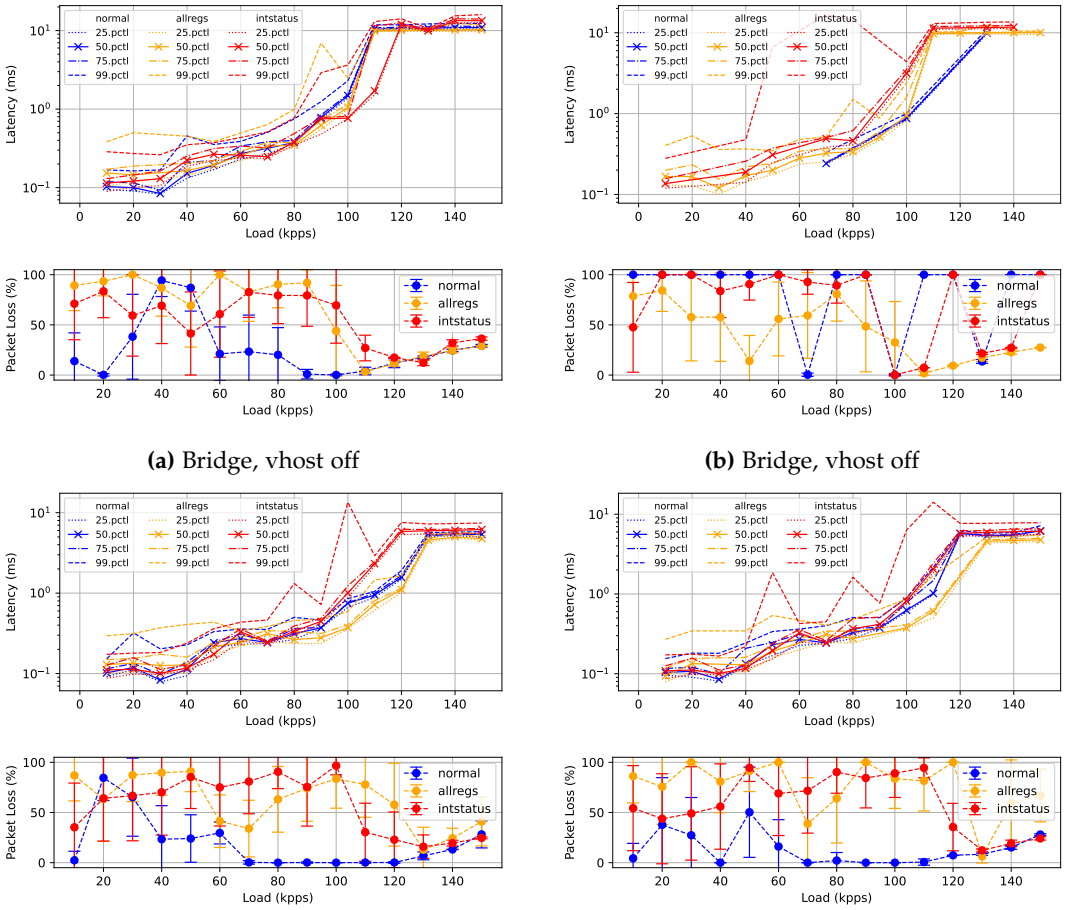
a VirtIO MMIO device, which is again suggests to be problematic by the fact, that packet loss below the load maximum is present in all cases here, it clearly seems like ioregionfd at least amplifies this issue. What we should mention here, is that we got the impression during our experiments, that we get less packet loss in long contiguous measurement runs, than in shorter bursts of measurements. We try to counter this by doing warm-up runs before the actual measurements, but this did not seem to have a significant effect. While we are not sure if MoonGen's `l2-load-latency.lua` script is the best tool to measure packet loss, we also cannot disregard these findings. To see if and how much this problem might influence the real world performance, we want to turn to the throughput next. While latency is usually measured with small packets, it is the opposite for the throughput. We therefore first see if the packet size affects the results we just saw.

6.2.2.6. Large Packets

Figure 6.9 compares the load latency and packet loss behavior for the same three cases than before: ioregionfd disabled, ioregionfd applied to just the `interrupt status` register, and ioregionfd applied to all registers that work, so all but `queue notify` and `interrupt acknowledge`. The important difference is, that the packet size is now not 64 B, but 1024 B. The figure again has the bridge plots on top and the MacVTap plots on the bottom, vhost is disabled on the left and enabled on the right.

In case of the bridge with vhost disabled, we only see three values for the latency without ioregionfd. For all the other cases, the packet loss was 100%, and therefore obviously did not leave any results. With nothing to compare, we ignore this case. The case of the bridge with vhost enabled shows a very similar behavior for ioregionfd disabled and applied to all registers, just that the 99th percentile latency is a bit higher with ioregionfd. With ioregionfd just applied to the interrupt status register, the average is lower around the maximum load, which also shifts 10 kpps up. The 99th percentile latency however is a lot higher in this area. For the MacVTap, we see similar curves for both vhost settings. Towards the load maximum the three curves split up, with ioregionfd applied to interrupt status being the highest, ioregionfd disabled in the middle, and ioregionfd applied to all registers the lowest. The larger gap in the vhost enabled case is likely the result of the missing measurement at 120 kpps for ioregionfd applied to interrupt status. When comparing these plots to the previous ones with the small packets, we see that we already had this split before, it just got wider with the larger packets, suggesting that the packet size intensifies the effects of ioregionfd. The trend of higher packet loss with ioregionfd also reappears however. There we again have to take the latency results with a grain of salt, as the packet loss could very well influence the latency results as well.

Just measuring latency for larger packets is however not enough to clearly measure throughput. While latency sets the upper bound for throughput, there are still other factors that can reduce the actual achievable throughput. We therefore now measure throughput, with the iPerf3 tool [139].



(a) Bridge, vhost off

(b) Bridge, vhost off

(c) Bridge, vhost off

(d) Bridge, vhost off

Figure 6.9.: Load latency and packet loss comparison for ioregionfd on and off, Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 1024 B, bus type is MMIO, and reflector is XDP.

6.2.2.7. Throughput

Just as for the latency measurements, we take the throughput measurements between the LoadGen machine and the Guest, see figure 6.2, with an iPerf3 server running on the guest, and an iPerf3 client running on the LoadGen machine. We take measurements for the same three ioregionfd configurations as before, ioregionfd disabled, ioregionfd applied to all possible registers, and ioregionfd applied to interrupt status only. Despite switching between bridge and MacVTap, as well as vhost on and off, we now also take the queue size as parameter, and measure for default size of 256 as well as 1024. The results are shown in table 6.1.

Note that all results are averages of five separate runs. For the TX (transmit) measurements, we use iPerf3 in reverse mode. Also note that the Retr column shows the

Interface / Vhost / Queue Size	ioregionfd					
	off		interrupt status		all registers	
	RX Mbps / Retr	TX Mbps / Retr	RX Mbps / Retr	TX Mbps / Retr	RX Mbps / Retr	TX Mbps / Retr
bridge/off/256	5046 / 21	9184 / 6037	4070 / 9	8960 / 4048	3698 / 97	8774 / 4277
bridge/off/1024	5026 / 118	9252 / 5343	4282 / 31	8640 / 3590	4024 / 5	8816 / 3201
bridge/on/256	5068 / 55	9168 / 5553	3946 / 180	8162 / 3471	3904 / 5	8858 / 3445
bridge/on/1024	5072 / 78	9106 / 5925	4354 / 10	8996 / 3664	3382 / 0	8752 / 2824
macvtap/off/256	5034 / 0	9270 / 2137	3444 / 6	8722 / 3039	2406 / 0	8296 / 2286
macvtap/off/1024	5086 / 10	9212 / 4896	2598 / 0	8716 / 3440	2434 / 0	8156 / 1490
macvtap/on/256	5080 / 18	9226 / 6082	2552 / 0	8692 / 1973	2410 / 0	8514 / 2238
macvtap/on/1024	5064 / 3	9168 / 6371	2502 / 0	8560 / 2478	2374 / 0	8388 / 2981

Table 6.1.: iPerf3 throughputs (Mbps) and retransmissions (Retr) results for different virtio-net-device configurations, including the three ioregionfd settings: off, applied to interrupt status only, and applied to all registers.

number of retransmissions. We can see that the NIC delivers almost the same performance for all cases, when ioregionfd is turned off: about 5000 Mbit/s for receival with few retransmissions, and about 9000 Mbit/s for transmission with a couple thousand retransmissions. We suspect that the fact that there are different systems, NICs and drivers on the receiving end for the RX and TX case (a 40 core machine with a 10G physical NIC in the TX case, and a 4 vCPU VM with a virtio NIC in the RX case), to be the reason why we see a stronger throughput for the TX case The guest and it's VirtIO NIC just can keep up with the baremetal system. This is however counter-intuitive for the retransmission, which behave the other way around. We think that while LoadGen does not throttle and buys a higher throughput at the expense of retransmissions, the Guest rather drops in speed for a cleaner transmission. The asymmetry in RX and TX throughput and retransmissions repeats for the other ioregionfd settings as well. The difference we do see is that with ioregionfd applied we only get roughly 8500 Mbit/s for transmissions, and 4000 Mbit/s for receival with the bridge and mostly just 2500 Mbit/s for receival with MacVTap. This is a clear performance drop with ioregionfd, more for the MacVTap than for the bridge, and more when all registers are handled by ioregionfd than just the interrupt status register. For the retransmissions on the other hand we get often none on receival, and also less on transmission than without ioregionfd. While this contradicts the trend of higher packet loss with ioregionfd, we saw in our latency measurements, thus might suggest that ioregionfd did not cause the packet loss, we think that this is simply a result of the lower throughput. Other parts of the setup are less likely to be overloaded and cause less packet drops. We should also mention, that in contrast to our latency measurements, we did not see packet loss during these iPerf3 measurements. We suspect that the TCP stack configured the transmission to avoid it.

6.2.3. Other Devices

To test whether out code also works for other devices as designed, we now also look at two more NICs and a storage device.

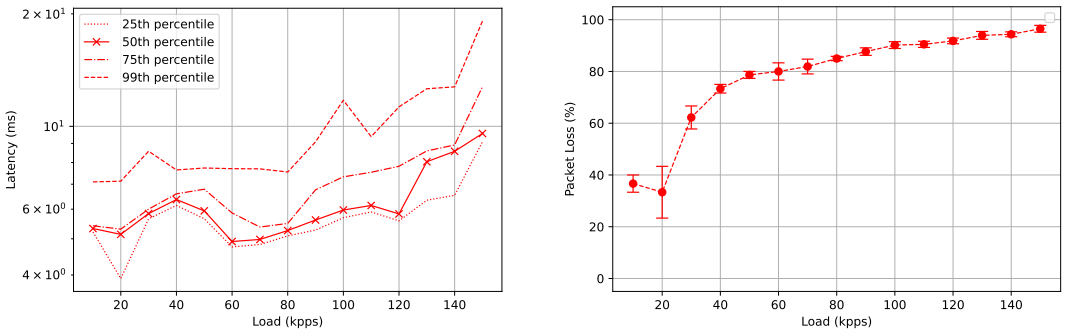
6.2.3.1. Emulated E1000 NIC

The first device we consider is QEMU's e1000 device. It is an emulated PCI NIC, which means we could test ioregionfd on the PCI bus, and thus might avoid the packet loss issues we experience with the VirtIO MMIO bus and also test ioregionfd with the MoonGen reflector. This way we might be able to rule out certain possible causes for the packet loss.

While we do manage to apply our code to the e1000 device with ease, we find that we could not get connectivity with the NIC even without ioregionfd. Because this appears to be a QEMU bug, we decide to look for alternatives.

6.2.3.2. Emulated RTL8139 NIC

Another emulated PCI NIC is QEMU's rtl8139 device. As for the e1000 we were able to quickly apply our code to this device. While binding the card to DPDK worked, running the MoonGen reflector program on the card results in an error. DPDK appears to not being able to find the NIC's queues. We therefore turn to the XDP reflector again, which works fine. We did expect the performance of the rtl8139 to be lower than for VirtIO NICs, especially since it is an emulated device, but we are still a little surprised to find out how low when looking at following results. Figure 6.10 shows the load latency and packet loss for one particular measurement.



(a) Load latency

(b) Packet loss

Figure 6.10.: Load latency and packet loss for the rtl8139 device, with a machine type pc, a bridge device, vhost off, ioregionfd off, XDP as reflector and a packet size of 64 B.

In this case the machine type is pc, the device is a bridge, vhost is off, ioregionfd is off, XDP is used as reflector and the packet size is 60 B. While the load latency is with over 5 ms already quite high at the low end of the plot, the packet loss reveals that the card is already overloaded at 20 kpps. When trying to measure this low in more detail, we find that MoonGen is apparently not able to accurately limit the rate, making measurements on this scale unreliable. In conjunction with the questionable usefulness of minor latency optimizations on such a low performing network device, we decide to not further investigate this device.

6.2.3.3. VirtIO Block Device

We already mentioned that applying ioregionfd to VirtIO's bus frontend means that all devices using the bus could profit from it. This made a test of the `virtio-blk-device` (VirtIO MMIO block device) an obvious choice. We intended to run FIO (Flexible I/O Tester) [117] on it. Unfortunately, we experience regular host crashes during these tests, even without ioregionfd. We suspect that these problems are hardware-related, which might also render potential results from such tests unreliable. With not enough time to migrate to another test system, we have to leave it at that.

7. Related Work

7.1. I/O Performance Issues in Virtual Machines

Rizzo et. al. introduce VALE [140], a virtual switched local ethernet for virtual machines to outperform up until then existing virtual switches and thus give operators of virtual machines an easy software solution for high performance networking in comparison to PCI passthrough for example. They later use this tool in later work [141] to demonstrate that packet rates beyond 1 Mpps are achievable within virtual machines with only minor modifications on device drivers, hypervisors and the host's switch. They furthermore show that the emulation of conventional NICs is capable of such rates without major changes to the device model.

Chen et. al. [142] argue that ARM-based virtualization is a promising foundation for future cloud platforms to decrease power consumption. They analysed the network performance of such virtual machines and observed that inefficient interrupt handling leads to major penalties that decrease ARMs attractiveness for cloud applications. They therefore implement a multi-channel network packet delivery via virtual MSI notification and show that this can improve the throughput of concurrent requests by up to 50% in certain scenarios.

Cheng et. al. [138] discuss the impact of network jitter in cloud environments and analyse the different causes. They propose two strategies to mitigate it: grouping different types of I/O on different CPUs for a better I/O type aware scheduling, as well as a stricter closed-loop feedback control on network resource consumption to smooth out network latency. They implement their approach in Xen and Linux to show the benefit of these techniques.

Lei et. all. [137] discuss the influence of the VirtIO ring buffer size on network performance. They show how to configure VirtIO devices with QEMU to have larger queues and how this can improve throughput and reduce packet loss.

7.2. Network Performance Measurements

Many tools exist to measure network performance. iPerf and its successor iPerf3 [139, 143] are very popular tools for throughput measurements. iPerf or iPerf3 have a client and server mode. Usually the client is the one sending traffic to the server, but iPerf3 also supports the opposite when run in reverse mode. iPerf/iPerf3 generate either TCP, UDP or SCTP traffic for their measurements.

netperf [144, 145] is a tool developed by Hewlett Packard for both unidirectional

throughput and end-to-end latency measurements. It support TCP, UDP DLPI, Unix Domain Sockets and SCTP. PROX (Packet pROcessing eXecution Engine) [146, 147] is a tool developed by Intel specifically for performance characterization of NFVI (Network Function Virtualization Infrastructure). IXIA [148] is a proprietary hardware test and simulation platform used for example by different NIC manufacturers, like Intel, Mellanox and Broadcom, for their DPDK performance reports [149]. TRex [150–154] is a DPDK-based open source traffic generator, that provides low-cost, stateless and stateful generation of L3-7 traffic. It allows alteration of any packet field, hardware timestamping, and per stream/group statistics. By supporting many protocols it provides the means to simulate real-world traffic at scale. pktgen [155–158] is also a DPDK-based high-speed packet generator, but build into the Linux kernel. It can handle UDP, TCP, ARP, ICMP, GRE, MPLS and Queue-in-Queue, and can be remotely controlled over a TCP connection. It can be configured with Lua for repeatable test cases.

Emmerich et. al. developed the high-speed packet generator MoonGen [123]. Like pktgen it uses the DPDK packet processing framework to bind directly to the NIC's DMA buffers to from user space to achieve packet rates that can fully saturate 10 Gb/s links with minimum frame sizes and at the same time be compatible with most commodity NICs. It uses the LuaJIT compiler to build the packet generation, transmission and reception from user-written LUA scripts making it very flexible. It also offers highly accurate sub-microsecond latency measurements through the hardware timestamping capabilities of modern NICs. All this combined it allows the user to turn nearly any commodity NIC into a packet generator those performance characteristics would usually require expensive hardware solutions. In later work [66] the authors use their packet generator to do an extensive quantitative black-box analysis on software switches with focus on Open vSwitch.

7.3. File-Descriptor-based Kernel-User-Space Communication Optimizations

Libenzi et. al. [114] introduce eventfd into the Linux kernel. This mechanism allows to create an in-kernel object holding a counter, that can be altered and read directly from user space over a specific file descriptor, to implement a bidirectional notification protocol between kernel and user space without involving expensive context switches. Based on this Haskins et. al. [11, 12] later introduces irqfd and ioeventfd, which implement an efficient host-to-guest interrupt and guest-to-host notification mechanism in KVM. While irqfd is made for interrupts, ioeventfd optimizes asynchronous writes to I/O device registers.

ioregionfd is a more general mechanism for binding guest device memory to a file-descriptor-based communication channel between KVM and the device emulator. In the RFC (request for comments) [14] Hajnoczi et. al. argue that current techniques in KVM to dispatch MMIO/PIO accesses have multiple shortcomings like performance penalties, and therefore propose this new mechanism for a more direct dispatch to the

device access handlers.

In a blog post [74] the author gives a high-level introduction to ioregionfd and later delivers a first implementation for the feature in the RFC [75].

7.4. Potential Use Cases for ioregionfd

Any emulated or para-virtualized device that is regularly accessed via MMIO might benefit from ioregionfd. While older emulated devices like the RTL8139 NIC [53, 54] or the E1000 NIC [51, 52] are probably most in need due to their rather low performance to begin with, the VirtIO [48, 118, 119] device family is a very interesting candidate due to its popularity. There are however also other use cases that might benefit from ioregionfd.

Both Stecklina et. al. [159] and Shi et. al. [160] critique that the purely performance-focussed monolithic architecture of hypervisors opens up large attack surfaces for example through I/O devices and puts the security of core modules in jeopardy. Both works propose solutions that secure hypervisors through adequate decomposition of their functionality, and proper isolation of the resulting parts. Out-of-hypervisor solutions like these also present potential use cases for further performance improvements through ioregionfd which can offer a direct communication channel between core hypervisor and the out-of-hypervisor devices.

A first example of this can be seen in RFC [79], where Ufimtseva implements a first application of the ioregionfd feature in QEMU, more specifically in QEMU's remote device. This device outsources the device backend to a separately run QEMU process for better isolation in context of security, while the frontend acts as a proxy using IPC (inter process communication) to pass on device accesses [77]. The author uses ioregionfd to allow for a direct communication between KVM and this backend, thus circumventing the device frontend in the main QEMU process to counter the performance penalties from the IPC and the context switches.

8. Conclusion

This thesis analysis how guest MMIO accesses can affect the performance of virtualized I/O devices on a theoretical level. We identify CPU mode switches between privileged and unprivileged mode of CPU virtualization extensions to give control back to the host, as well as context switches from host kernel to host user space to run the device's access handler, as the main performance bottlenecks. We then examine how the newly proposed KVM feature, ioregionfd, attempts to circumvent the context switch with a direct file-descriptor-based communication protocol between KVM and the device emulator. We use this knowledge to implement a proof-of-concept implementation applying ioregionfd to the VirtIO MMIO bus frontend in QEMU. We show how splitting the ioregionfd handler into a device specific callback function and a device agnostic core handler, which is called with function pointers to the original device access handlers, makes the implementation generic enough to be applicable to other QEMU devices with minor modifications. We also explain how ioregionfd's wire protocol can be debugged from both ends by the means of nested virtualization, with one GDB attached to the "virtual" host's QEMU process and the other to the "virtual" host's kernel from the outermost machine. We then evaluate our code on the VirtIO MMIO network device thoroughly taking into account the effect and interaction of several configuration parameters.

We found that an Intel 82599ES-based physical NIC delivers the expected performance by handling minimal packet sizes at 10 Gb/s, only showing a slight increase in the latency of the 99th percentile towards the end. We also saw that while DPDK is fully capable of operating with such high packet rates, more than 14 Mpps, XDP was only able to handle 1.5 Mpps in our setup. We found that VirtIO NICs do not even reach this packet rate by far, in our case they usually start losing packets at around 120 kpps. We could confirm that MacVTap interfaces often perform better, with sometimes multiple milliseconds lower latency and rarely a bit higher maximum packet rate in comparison to Linux bridges, but that there are exceptions where the opposite is true. While we could not produce any improvements with optimizations like the vhost protocol or an increased VirtIO queue size, we uncover random packet loss behavior for the VirtIO MMIO network device, maybe hinting to a bug in the VirtIO MMIO bus. Under these unfortunate circumstances we were unable to make out any performance improvements through the use of ioregionfd, but observe a general trend of even higher packet loss in its presence. We saw that this behavior is not limited to small packet sizes but also negatively impacts the maximum throughput. Because the root cause of the packet loss needs further research, we cannot not judge whether ioregionfd causes packet loss under certain conditions or if it simply amplified the pre-existing issue.

Even though our verdict on ioregionfd's potential benefit therefore remains inconclusive, we believe this to be a good foundation for further research. The VirtIO block device definitely deserves another look at, but there are surely more devices. The influence of VirtIO's queue size should also get more attention. A closer look at the VirtIO guest drivers could also prove to be interesting. Adding additional register accesses could be a way to cause the necessary MMIO load to properly measure the impact of ioregionfd. It could also help to better understand what registers should be handled by ioregionfd, if at all. I might also uncover the cause for VirtIO MMIO's random packet loss, which in our opinion is the most pressing matter. An interesting hint may be the fact, that we only observe the packet loss with MoonGen but we did not directly observe packet loss during our TCP-based iPerf3 tests, here more investigation is definitely needed.

A. Appendix

A.1. Autotest Config File

The following listing A.1 shows two sections of the autotest config file we used for our VirtIO measurements. It instructs autotest to run tests for all possible combinations of of the different parameters.

```
[virtio]
machines = pcvm, microvm
interfaces = bridge, macvtap
qemus = normal:/home/networkadmin/qemu_build
vhosts = false, true
ioregionfds = false
reflectors = moongen, xdp
rates = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150
size = 60
runtimes = 30
repetitions = 3
warmup = false
cooldown = false
accumulate = true
outputdir = ./out/virtio

[virtio-ioregionfd-interrupt-status]
machines = microvm
interfaces = bridge, macvtap
qemus = intstatus:/home/networkadmin/qemu3_build
vhosts = false, true
ioregionfds = true
reflectors = xdp
rates = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150
size = 60
runtimes = 30
repetitions = 3
warmup = false
cooldown = false
accumulate = true
outputdir = ./out/virtio
```

Listing A.1: Example of an autotest config file for measurements on the the virtio network devices.

A.2. Packet Loss with virtio-net-device

The following listing A.2 shows and example of MoonGen's `l2-load-latency.lua` script's output showcasing seemingly random full packet loss with the `virtio-net-device`.

```
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.02 Mpps, 11 Mbit/s (15 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 26 Mbit/s (34 Mbit/s with framing)
[Device: id=1] RX: 0.03 Mpps, 17 Mbit/s (22 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 26 Mbit/s (34 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.05 Mpps, 27 Mbit/s (35 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 27 Mbit/s (35 Mbit/s with framing)
[Device: id=1] RX: 0.05 Mpps, 27 Mbit/s (35 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 27 Mbit/s (35 Mbit/s with framing)
[Device: id=1] RX: 0.02 Mpps, 13 Mbit/s (16 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 26 Mbit/s (34 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.02 Mpps, 8 Mbit/s (11 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.05 Mpps, 26 Mbit/s (35 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 27 Mbit/s (35 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
[Device: id=1] RX: 0.00 Mpps, 0 Mbit/s (0 Mbit/s with framing)
[Device: id=1] TX: 0.05 Mpps, 25 Mbit/s (33 Mbit/s with framing)
```

Listing A.2: Example of MoonGen output showing random packet loss with the virtio-net-device.

List of Figures

List of Tables

Bibliography

[1] G. A. A. Santana, *Data Center Virtualization Fundamentals: Understanding Techniques and Designs for Highly Efficient Data Centers with Cisco Nexus, UCS, MDS, and Beyond*, 1st ed. WebEx Communications, 2013.

[2] P. Padala, X. Zhu, Z. Wang, and S. Singhal, "Performance evaluation of virtualization technologies for server consolidation," *Technical Report HPL-2007-59R1, HP Laboratories*, 01 2007.

[3] W. Vogels, "Beyond server consolidation: Server consolidation helps companies improve resource utilization, but virtualization can help in other ways, too." *Queue*, vol. 6, no. 1, p. 20–26, 1 2008. [Online]. Available: https://doi.org/10.1145/1348583.1348590

[4] H. Lv, Y. Dong, J. Duan, and K. Tian, "Virtualization challenges: A view from server consolidation perspective," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 15–26. [Online]. Available: https://doi.org/10.1145/2151024.2151030

[5] A. Varasteh and M. Goudarzi, "Server consolidation techniques in virtualized data centers: A survey," *IEEE Systems Journal*, vol. 11, no. 2, pp. 772–783, 2017.

[6] Y. Xing and Y. Zhan, "Virtualization and cloud computing," in *Future Wireless Networks and Information Systems*, Y. Zhang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 305–312.

[7] Geeta and S. Prakash, "Role of virtualization techniques in cloud computing environment," in *Advances in Computer Communication and Computational Sciences*, S. K. Bhatia, S. Tiwari, K. K. Mishra, and M. C. Trivedi, Eds. Singapore: Springer Singapore, 2019, pp. 439–450.

[8] N. Jain and S. Choudhary, "Overview of virtualization in cloud computing," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, 2016, pp. 1–4.

[9] A. C. Aaqib Rashid, "Virtualization and its role in cloud computing environment," *International Journal of Computer Sciences and Engineering*, vol. 7, pp. 1131–1136, 4 2019. [Online]. Available: https://www.ijcseonline.org/full_paper_view.php?paper_id=4177

[10] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 36–47, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548911000559

[11] A. K. Gregory Haskins. (2009) Kvm: irqfd. [Online]. Available: https://github.com/torvalds/linux/commit/721eecbf4fe995ca94a9edec0c9843b1cc0eaaf3 (Accessed 2022-09-01).

[12] A. K. Gregory Haskins, Michaerl S. Tsirkin. (2009) Kvm: add ioeventfd support. [Online]. Available: https://github.com/torvalds/linux/commit/d34e6b175e61821026893ec5298cc8e7558df43a (Accessed 2022-09-01).

[13] S. H. et. al. (2011) virtio: Use ioeventfd for virtqueue notify. [Online]. Available: https://lore.kernel.org/all/AANLkTi=JbDFZyjLgBePSMe5g2UDpGX6MVXXez10wFE8W@mail.gmail.com/T/ (Accessed 2022-09-01).

[14] S. Hajnoczi. (2020) Proposal for MMIO/PIO dispatch file descriptors (ioregionfd). [Online]. Available: https://www.spinics.net/lists/kvm/msg208139.html (Accessed 2022-03-19).

[15] J. D. Mark A. Peloquin. (2018) Kvm network performance - best practices and tuning recommendations. IBM Corporation. [Online]. Available: https://www.ibm.com/downloads/cas/ZVJGQX8E (Accessed 2022-09-10).

[16] R. van Riel. (2011) Kvm performance optimization internals. Red Hat Inc. [Online]. Available: https://www.surriel.com/system/files/riel_t_1620_kvm_performance.pdf (Accessed 2022-09-10).

[17] W. Jiang, Y. Zhou, Y. Cui, W. Feng, Y. Chen, Y. Shi, and Q. Wu, "Cfs optimizations to kvm threads on multi-core environment," in *2009 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 348–354.

[18] Q. Hao and Z. Guo, "Optimization of kvm network based on cpu affinity on multi-cores," in *Information Technology, Computer Engineering and Management Sciences, International Conference of*, vol. 4. Los Alamitos, CA, USA: IEEE Computer Society, 9 2011, pp. 347–351. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICM.2011.330

[19] (2007) "intel® virtualization technology (vt) in converged application platforms". [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-tech-converged-application-platforms-paper.pdf (Accessed 2022-09-02).

[20] I. Corporation, *Intel Virtualization Technology for Directed I/O*, 2022. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf (Accessed 2022-09-02).

[21] "A performance comparison of hypervisors," 2007. [Online]. Available: https://www.vmware.com/pdf/hypervisor_performance.pdf (Accessed 2022-09-02).

[22] O. A. Keith Adams, "A comparison of software and hardware techniques for x86 virtualization," 2006. [Online]. Available: https://www.vmware.com/pdf/asplos235_adams.pdf (Accessed 2022-09-02).

[23] N. Bhatia, "Performance evaluation of intel ept hardware assist," 2009. [Online]. Available: https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf (Accessed 2022-09-02).

[24] J. Fisher-Ogden, ""hardware support for efficient virtualization"," *University of California, San Diego, Tech. Rep*, vol. 12, 2006. [Online]. Available: https://course.ece.cmu.edu/~ece845/sp16/docs/hardwareVirt.pdf (Accessed 2022-09-02).

[25] R. Ganesan, Y. Murarka, S. Sarkar, and K. Frey, "Empirical study of performance benefits of hardware assisted virtualization," in *Proceedings of the 6th ACM India Computing Convention*, ser. Compute '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2522548.2522598

[26] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization," vol. 10, 08 2006.

[27] A. Faravelon, O. Gruber, and F. Pétrot, "Optimizing memory access performance using hardware assisted virtualization in retargetable dynamic binary translation," in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 40–46.

[28] H. N. Palit, X. Li, S. Lu, L. C. Larsen, and J. A. Setia, "Evaluating hardware-assisted virtualization for deploying hpc-as-a-service," in *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, ser. VTDC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 11–20. [Online]. Available: https://doi.org/10.1145/2465829.2465833

[29] (2022) About Nested Virtualization. Google LLC. [Online]. Available: https://cloud.google.com/compute/docs/instances/nested-virtualization/overview (Accessed 2022-08-21).

[30] (2021) Getting started with KVM. IBM Corporation. [Online]. Available: https://www.ibm.com/docs/en/cic/1.1.3?topic=SSLL2F_1.1.3/com.ibm.cloudin.doc/overview/Getting_started_tutorial.html (Accessed 2022-08-21).

[31] (2022) Oracle Virtualization. Oracle. [Online]. Available: https://www.oracle.com/virtualization/ (Accessed 2022-08-21).

[32] S. Sharwood. (2017) Aws adopts home-brewed kvm as new hypervisor. [Online]. Available: https://www.theregister.com/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/ (Accessed 2022-08-21).

[33] Openstack. (2021) Most commonly used OpenStack compute (Nova) hypervisors worldwide, as of 2020. [Online]. Available: https://www.statista.com/statistics/1109443/worldwide-openstack-hypervisors/ (Accessed 2022-08-21).

[34] C. Aker. (2015) Linode turns 12! Here's some KVM! Linode LLC. [Online]. Available: https://www.linode.com/blog/linode/linode-turns-12-heres-some-kvm/ (Accessed 2022-09-10).

[35] C. Aker. (2016) KVM Update. Linode LLC. [Online]. Available: https://www.linode.com/blog/linux/kvm-update/ (Accessed 2022-09-10).

[36] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *J. Parallel Distrib. Comput.*, vol. 72, no. 11, p. 1471–1480, 11 2012. [Online]. Available: https://doi.org/10.1016/j.jpdc.2012.01.020

[37] F. W. Maximilian Fischer, "Survey on sr-iov performance," 2022. [Online]. Available: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2022-01-1/NET-2022-01-1_09.pdf (Accessed 2021-09-10).

[38] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support," pp. 1–12, 2010.

[39] C. Robison, *Configure SR-IOV Network Virtual Functions in Linux* KVM**, Intel Corporation, 2017. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/configure-sr-iov-network-virtual-functions-in-linux-kvm.html (Accessed 2021-09-10).

[40] D. D. Yu Zhao, *PCI Express I/O Virtualization Howto*, 2009. [Online]. Available: https://docs.kernel.org/PCI/pci-iov-howto.html (Accessed 2021-09-10).

[41] W. Han. (2015) Live migration with sr-iov pass-through. Huawei Technologies Co., Ltd. [Online]. Available: http://www.linux-kvm.org/images/9/9a/03x07-Juniper-Weidong_Han-LiveMigrationWithSR-IOVPass-through.pdf (Accessed 2021-09-10).

[42] A. S. Chavan, A. S. Varal, V. S. Bhende, and M. Thalor, "Experimental analysis of dedicated gpu in virtual framework using vgpu," in *2021 International Conference on Emerging Smart Computing and Informatics (ESCI)*, 2021, pp. 483–489.

[43] A. Garg, P. Kulkarni, U. Kurkure, H. Sivaraman, and L. Vu, "Empirical analysis of hardware-assisted gpu virtualization," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 395–405.

[44] R. Leira, G. Julián-Moreno, I. González, F. J. Gómez-Arribas, and J. E. López de Vergara, "Performance assessment of 40 gbit/s off-the-shelf network cards for virtual network probes in 5g networks," *Computer Networks*, vol. 152, pp. 133–143, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138912861930101X

[45] *Enhanced Networking on Linux*, Amazon Web Services, 2022. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html (Accessed 2021-09-10).

[46] *Set up SR-IOV networking*, Google Cloud, 2022. [Online]. Available: https://cloud.google.com/anthos/clusters/docs/bare-metal/latest/how-to/sriov (Accessed 2021-09-10).

[47] *SR-IOV backet networks*, IBM Corporation, 2022. [Online]. Available: https://www.ibm.com/docs/en/powervc-cloud/1.4.4?topic=networks-sr-iov-backed (Accessed 2021-09-10).

[48] M. S. T. et. al., *Virtual I/O Device (VIRTIO) Version 1.1*, 2019. [Online]. Available: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf (Accessed 2022-09-02).

[49] E. P. Martín. (2020) Virtio devices and drivers overview: The headjack and the phone. [Online]. Available: https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone (Accessed 2022-04-02).

[50] E. P. Martín. (2019) Deep dive into virtio-networking and vhost-net. [Online]. Available: https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net (Accessed 2022-09-02).

[51] *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual*, Intel Corporation, 2012. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/pcie-gbe-controllers-open-source-manual.pdf (Accessed 2021-09-10).

[52] T. Q. P. Developers. (2022) qemu/hw/net/e1000.c at master · qemu/qemu · github. Qumranet. [Online]. Available: https://github.com/qemu/qemu/blob/master/hw/net/e1000.c (Accessed 2021-09-10).

[53] *Realtek RTL8139D RTL8139DL RTL8139D-LF RTL8139DL-LF RTL8139D-GR RTL8139DL-GRSINGLE-CHIP MULTI-FUNCTION 10/100Mbps ETHERNET CONTROLLER WITH POWER MANAGEMENT DATASHEET*, Realtek Semiconductor, 2005. [Online]. Available: http://realtek.info/pdf/rtl8139d.pdf (Accessed 2021-09-10).

[54] T. Q. P. Developers. (2022) qemu/hw/net/rtl8139.c at master · qemu/qemu · github. Qumranet. [Online]. Available: https://github.com/qemu/qemu/blob/master/hw/net/rtl8139.c (Accessed 2021-09-10).

[55] E. P. Martín. (2020) Virtqueues and virtio ring: How the data travels. [Online]. Available: https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels (Accessed 2022-09-02).

[56] A. K. et. al. (2018) Virtio uses dma api for all devices. [Online]. Available: https://patchwork.ozlabs.org/project/linuxppc-dev/cover/20180720035941.6844-1-khandual@linux.vnet.ibm.com/ (Accessed 2022-09-02).

[57] Peterx. (2022) Features/vt-d. [Online]. Available: https://wiki.qemu.org/Features/VT-d (Accessed 2022-09-02).

[58] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed. USA: Prentice Hall Press, 2012.

[59] M. Rafiquzzaman, *Fundamentals of Digital Logic and Microcontrollers*, 6th ed. Wiley Publishing, 2014.

[60] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[61] Terenceli. (2018) Kvm mmio implementation. [Online]. Available: https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2018/09/03/kvm-mmio (Accessed 2022-09-02).

[62] walkero. (2022) Mmio simulation principle. [Online]. Available: https://programmer.ink/think/mmio-simulation-principle.html (Accessed 2022-09-02).

[63] L. T. et. al. (2022) Linux kvm handle_ept_misconfig. [Online]. Available: https://github.com/torvalds/linux/blob/5995497296ade7716c8e70899e02235f2b6d9f5d/arch/x86/kvm/vmx/vmx.c#L5675-L5694 (Accessed 2022-09-02).

[64] L. T. et. al. (2022) Linux kvm vmx_handle_exit. [Online]. Available: https://github.com/torvalds/linux/blob/5995497296ade7716c8e70899e02235f2b6d9f5d/arch/x86/kvm/vmx/vmx.c#L6283-L6483 (Accessed 2022-09-02).

[65] QEMU Project Developers. (2022) Qemu kvm_cpu_exec. [Online]. Available: https://github.com/qemu/qemu/blob/

61fd710b8da8aedcea9b4f197283dc38638e4b60/accel/kvm/kvm-all.c#L2869-L3044 (Accessed 2022-09-02).

[66] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of virtual switching with open vswitch: A quantitative analysis," *Journal of Network and Systems Management*, vol. 26, 04 2018. [Online]. Available: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/Jnasm2017.pdf (Accessed 2022-08-21).

[67] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, ser. MMSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 35–46. [Online]. Available: https://doi.org/10.1145/1730836.1730842

[68] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, "Towards high-quality i/o virtualization," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1534530.1534547

[69] D. Ghoshal, R. S. Canon, and L. Ramakrishnan, "I/o performance of virtualized cloud environments," in *Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds*, ser. DataCloud-SC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 71–80. [Online]. Available: https://doi.org/10.1145/2087522.2087535

[70] B. Hou, F. Chen, Z. Ou, R. Wang, and M. Mesnier, "Understanding i/o performance behaviors of cloud storage from a client's perspective," *ACM Trans. Storage*, vol. 13, no. 2, 5 2017. [Online]. Available: https://doi.org/10.1145/3078838

[71] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, 2017.

[72] B. Zhang, X. Wang, R. Lai, L. Yang, Y. Luo, X. Li, and Z. Wang, "A survey on i/o virtualization and optimization," in *2010 Fifth Annual ChinaGrid Conference*, 2010, pp. 117–123.

[73] P. Ivanovic and H. Richter, "Openstack cloud tuning for high performance computing," in *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, 2018, pp. 142–146.

[74] sbrksb. (2020) A brief introduction to the ioregionfd project. [Online]. Available: https://sbrksb.github.io/2020/12/10/intro.html (Accessed 2022-03-19).

[75] E. Afanasova. (2021) Introduce mmio/pio dispatch file descriptors (ioregionfd). [Online]. Available: https://lore.kernel.org/all/cover.1613828726.git.eafanasova@gmail.com/ (Accessed 2022-08-21).

[76] J. Thalheim and sbrksb. (2021) Comparing changes (linux peter/5.12.14-v0 with v5.12.14). [Online]. Available: https://github.com/vmuxIO/linux/compare/v5.12.14...vmuxIO:peter/5.12.14-v0 (Accessed 2022-03-19).

[77] *Multi-process QEMU*, The QEMU Project Developers, 2022. [Online]. Available: https://www.qemu.org/docs/master/system/multi-process.html (Accessed 2022-03-19).

[78] *Multi-process QEMU*, The QEMU Project Developers, 2022. [Online]. Available: https://www.qemu.org/docs/master/devel/multi-process.html (Accessed 2022-03-19).

[79] E. Ufimtseva. (2021) ioregionfd introduction. [Online]. Available: https://lists.nongnu.org/archive/html/qemu-devel/2022-02/msg01573.html (Accessed 2022-03-19).

[80] QEMU project developers, Elena Ufimtseva and Sandro-Alessio Gierens. (2022) vmuxio/qemu ioregionfd. [Online]. Available: https://github.com/vmuxIO/qemu/tree/ioregionfd (Accessed 2022-08-24).

[81] L. Null and J. Lobur, *The Essentials of Computer Organization And Architecture*. USA: Jones and Bartlett Publishers, Inc., 2006.

[82] D. T. Wang, *Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm*, 2005. [Online]. Available: https://user.eng.umd.edu/~blj/papers/thesis-PhD-wang--DRAM.pdf (Accessed 2021-09-08).

[83] M. Seaborn. (2015) How physical addresses map to rows and banks in dram. [Online]. Available: http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html (Accessed 2021-09-08).

[84] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[85] David. (2017) AMD Zen SOC. [Online]. Available: https://en.wikichip.org/wiki/File:zen_soc.png (Accessed 2022-08-20).

[86] M. Anumothu, "Design and analysis of dma controller for system on chip based applications," *International Journal of VLSI and Embedded Systems-IJVES*, vol. 07, pp. 1685–1690, 06 2016.

[87] *Supermicro H12SSW-IN/H12SSW-NT User's Manual*, Super Micro Computer, Inc., 2022. [Online]. Available: https://www.supermicro.com/manuals/motherboard/EPYC7000/MNL-2220.pdf (Accessed 2021-09-08).

[88] V. Moreno, "Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10gbps networks," Ph.D. dissertation, 09 2012.

[89] R. Williams, *Computer Systems Architecture: A Networking Approach*, ser. Computer Systems Architecture: A Networking Approach. Addison-Wesley, 2001, no. Bd. 1. [Online]. Available: https://books.google.de/books?id=P4cpAQAAMAAJ

[90] D. Serpanos and T. Wolf, *Architecture of Network Systems*, ser. ISSN. Elsevier Science, 2011. [Online]. Available: https://books.google.de/books?id=o7GhrZXM_ssC

[91] Y. Klimiankou, "Design and implementation of port-mapped io management subsystem and kernel interface for true microkernels on ia-32 processors," *Programming and Computer Software*, vol. 45, pp. 319–323, 11 2019.

[92] A. K. Jack Belzer, Albert G. Holzman, *Encyclopedia of Computer Science and Technology: Volume 10 - Linear and Matrix Algebra to Microorganisms: Computer-Assisted Identification*. CRC Press, 1978.

[93] K. Hong. (2020) Memory-mapped io vs port-mapped io. [Online]. Available: https://www.bogotobogo.com/Embedded/memory_mapped_io_vs_port_mapped_isolated_io.php (Accessed 2021-09-08).

[94] E. D. Reilly, *Memory-Mapped I/O*. GBR: John Wiley and Sons Ltd., 2003, p. 1152.

[95] A. Papagiannis, M. Marazakis, and A. Bilas, "Memory-mapped i/o on steroids," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 277–293. [Online]. Available: https://doi.org/10.1145/3447786.3456242

[96] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Boston, MA: Pearson, 2014.

[97] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[98] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis, "A case against (most) context switches," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 17–25. [Online]. Available: https://doi.org/10.1145/3458336.3465274

[99] (2016) "intel® 64 and ia-32 architectures software developer's manual volume 3c: System programming guide, part 3". Intel Corporation. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/

manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf (Accessed 2022-09-02).

[100] (2019) "amd64 architecture programmer's manual volume 2: System programming". Advanced Micro Devices, Inc. [Online]. Available: https://www.amd.com/system/files/TechDocs/24593.pdf (Accessed 2022-09-02).

[101] R. H. Inc. (2022) What is KVM? [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-KVM (Accessed 2022-08-21).

[102] R. et. al. (2011) QEMU-KVM-Buch / Grundlagen. [Online]. Available: http://qemu-buch.de/de/index.php?title=QEMU-KVM-Buch/_Grundlagen (Accessed 2022-08-21).

[103] T. Q. P. Developers, *About QEMU*, 2022. [Online]. Available: https://www.qemu.org/docs/master/about.html (Accessed 2022-08-21).

[104] olive.zhao. (2021) Comparison of KVM and XEN Technologies. [Online]. Available: https://forum.huawei.com/enterprise/en/comparison-of-kvm-and-xen-technologies/thread/773247-893 (Accessed 2022-08-21).

[105] *The memory API*, QEMU project developers, 2022. [Online]. Available: https://www.qemu.org/docs/master/devel/memory.html (Accessed 2022-03-19).

[106] J. et. al. (2021) Context switching. [Online]. Available: https://wiki.osdev.org/Context_Switching (Accessed 2022-09-02).

[107] F. M. David, J. C. Carlyle, and R. H. Campbell, "Context switch overheads for linux on arm platforms," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 3–es. [Online]. Available: https://doi.org/10.1145/1281700.1281703

[108] B. Ouni, C. Belleudy, S. Bilavarn, and E. Senn, "Embedded operating systems energy overhead," in *Proceedings of the 2011 Conference on Design & Architectures for Signal & Image Processing (DASIP)*, 2011, pp. 1–6.

[109] F. David, J. Carlyle, and R. Campbell, "Context switch overheads on mobile device platforms," pp. 2–2, 06 2007.

[110] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 2–es. [Online]. Available: https://doi.org/10.1145/1281700.1281702

[111] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," *SIGPLAN Not.*, vol. 26, no. 4, p. 75–84, 4 1991. [Online]. Available: https://doi.org/10.1145/106973.106982

[112] MaiZure. (2018) Evolution of the x86 context switch in linux. [Online]. Available: https://www.maizure.org/projects/evolution_x86_context_switch_linux/ (Accessed 2022-09-02).

[113] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. [Online]. Available: https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization

[114] L. T. Davide Libenzi, Andrew Morton. (2007) signal/timer/eventfd: eventfd core. [Online]. Available: https://github.com/torvalds/linux/commit/e1ad7468c77ddb94b0615d5f50fa255525fde0f0 (Accessed 2022-09-01).

[115] Linux Project Developers. (2021) eventfd(2). [Online]. Available: https://man7.org/linux/man-pages/man2/eventfd.2.html (Accessed 2022-09-01).

[116] QEMU project developers, Elena Ufimtseva and Sandro-Alessio Gierens. (2022) vmuxio/qemu ioregionfd. [Online]. Available: https://github.com/vmuxIO/qemu/tree/ioregionfd (Accessed 2022-08-24).

[117] J. A. et. al., *fio - Flexible I/O tester rev. 3.30*, 2022. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html (Accessed 2022-09-12).

[118] S. et. al. (2021) Virtio. [Online]. Available: https://wiki.osdev.org/Virtio (Accessed 2022-09-02).

[119] (2016) Virtio. Red Hat, Inc. [Online]. Available: https://www.linux-kvm.org/page/Virtio (Accessed 2022-09-02).

[120] Q. Developers". (2022) 'microvm' virtual platform (microvm). [Online]. Available: https://qemu.readthedocs.io/en/latest/system/i386/microvm.html (Accessed 2022-09-02).

[121] Sandro-Alessio Gierens. (2022) vmuxio/qemu ioregionfd. [Online]. Available: https://github.com/vmuxIO/qemu/compare/ioregfd-intro...vmuxIO:qemu:ioregionfd (Accessed 2022-08-24).

[122] QEMU project developers. (2022) vmuxio/qemu v6.2.0-rc4. [Online]. Available: https://github.com/vmuxIO/qemu/tree/v6.2.0-rc4 (Accessed 2022-08-24).

[123] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–287. [Online]. Available: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/MoonGen_IMC2015.pdf (Accessed 2022-08-21).

[124] DPDK Developers, *DPDK Documentation*, 2022. [Online]. Available: https://core.dpdk.org/doc/ (Accessed 2022-08-30).

[125] S.-A. Gierens. (2022) xdp-reflector. [Online]. Available: https://github.com/gierens/xdp-reflector (Accessed 2022-08-30).

[126] XDP Developers, *XDP Documentation*, 2022. [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html (Accessed 2022-08-30).

[127] Linux Project Developers. (2021) ip-link(8). [Online]. Available: https://man7.org/linux/man-pages/man8/ip-link.8.html (Accessed 2022-09-01).

[128] s. Sandro-Alessio Gierens, Jörg Thalheim. (2022) linux ioregionfd. [Online]. Available: https://github.com/vmuxIO/linux/tree/ioregionfd (Accessed 2022-08-30).

[129] Sandro-Alessio Gierens, The QEMU Project Developers. (2022) qemu ioregionfd. [Online]. Available: https://github.com/vmuxIO/qemu/tree/ioregionfd (Accessed 2022-08-30).

[130] S.-A. Gierens. (2022) autotest. [Online]. Available: https://github.com/vmuxIO/autotest/ (Accessed 2022-08-30).

[131] P. Abine. (2018) Achieving high-performance, low-latency networking with xdp: Part i. [Online]. Available: https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1 (Accessed 2022-09-10).

[132] P. Abine. (2018) Using express data path (xdp) maps in rhel 8: Part 2. [Online]. Available: https://developers.redhat.com/blog/2018/12/17/using-xdp-maps-rhel8 (Accessed 2022-09-10).

[133] J. D. Brouer. (2017) Xdp infrastructure development. [Online]. Available: http://vger.kernel.org/netconf2017_files/XDP_devel_update_NetConf2017_Seoul.pdf (Accessed 2022-09-10).

[134] P. R. X. C. e. a. Eduardo Freitas, Assis T. de Oliveira Filho, "Takeaways from an experimental evaluation of xdp and dpdk under a cloud computing environment," 2022.

[135] D. B. Johan Almbladh. (2022) torvalds/linux: xdp: Fix spurious packet loss in generic xdp tx path. [Online]. Available: https://github.com/torvalds/linux/commit/1fd6e5675336daf4747940b4285e84b0c114ae32 (Accessed 2022-09-12).

[136] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5g qos: Impact of security functions on latency," 04 2020, pp. 1–9.

[137] Y. Lei. (2018) Vhost/virtio zero-packet-loss configuration optimization. Intel Corporation. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/vhostvirtio-zero-packet-loss-configuration-optimization.html (Accessed 2022-09-12).

[138] L. Cheng, C.-L. Wang, and S. Di, "Defeating network jitter for virtual machines," in *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, 2011, pp. 65–72.

[139] R. McMahon, B. Kaushik, and e. a. Tim Auckland. (2022) iperf - the ultimate speed test tool for tcp, udp and sctp. [Online]. Available: https://iperf.fr/ (Accessed 2022-09-12).

[140] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 61–72. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.677.8864&rep=rep1&type=pdf (Accessed 2022-08-21).

[141] L. Rizzo, G. Lettieri, and V. Maffione, "Speeding up packet i/o in virtual machines," in *Architectures for Networking and Communications Systems*, 2013, pp. 47–58. [Online]. Available: http://info.iet.unipi.it/~luigi/papers/20130520-rizzo-vm.pdf (Accessed 2022-08-21).

[142] B. Chen, "Optimizing network i/o virtualization for scale-out processor," in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2018, pp. 1628–1635. [Online]. Available: https://ieeexplore.ieee.org/document/8560257 (Accessed 2022-08-21).

[143] B. A. M. et. al. (2022) esnet/iperf. [Online]. Available: https://github.com/esnet/iperf (Accessed 2022-09-12).

[144] (2021) The netperf homepage. Hewlett-Packard. [Online]. Available: https://hewlettpackard.github.io/netperf/ (Accessed 2021-09-12).

[145] Hewlett-Packard. (2021) Hewlettpackard/netperf. [Online]. Available: https://github.com/HewlettPackard/netperf (Accessed 2021-09-12).

[146] Y. Kylulin, L. Provoost, and P. Torre, *Packet pROcessing eXecution Engine (PROX) - Performance Characterization for NFVI User Guide*, 2020. [Online]. Available: https://builders.intel.com/docs/networkbuilders/packet-processing-execution-engine-prox-performance-characterization-for-nfvi-user-guide.pdf (Accessed 2021-09-12).

[147] treyad and A. Komarov. (2019) nvf-crucio/prox. [Online]. Available: https://github.com/nvf-crucio/PROX (Accessed 2021-09-12).

[148] K. Technologies. (2020) Ixia, a keysight business, is now keysight. [Online]. Available: https://www.keysight.com/us/en/cmp/2020/network-visibility-network-test.html (Accessed 2021-09-12).

[149] (2022) Dpdk performance reports. DPDK Project. [Online]. Available: https://core.dpdk.org/perf-reports/ (Accessed 2021-09-12).

[150] (2022) Trex realistic traffic generator. TRex Team, Cisco Systems. [Online]. Available: https://trex-tgn.cisco.com/ (Accessed 2021-09-12).

[151] (2022) Trex. TRex Team, Cisco Systems. [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_manual.html (Accessed 2021-09-12).

[152] (2022) Trex stateless supoort. TRex Team, Cisco Systems. [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_stateless.html (Accessed 2021-09-12).

[153] C. S. TRex Team. (2022) cisco-system-traffic-generator. [Online]. Available: https://github.com/cisco-system-traffic-generator (Accessed 2021-09-12).

[154] *Practive with TRex*, DPDK Project, 2017. [Online]. Available: https://doc.dpdk.org/dts/gsg/usr_guide/trex.html (Accessed 2021-09-12).

[155] K. Wiles, *The Pktgen Application*, 2019. [Online]. Available: https://pktgen-dpdk.readthedocs.io/en/latest/ (Accessed 2021-09-12).

[156] *HOWTO for the linux packet generator*, The kernel development community, 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/pktgen.html (Accessed 2021-09-12).

[157] R. Olsson, "pktgen the linux packet generator," 2005. [Online]. Available: http://www.cs.columbia.edu/~nahum/w6998/papers/ols2005v2-pktgen.pdf (Accessed 2021-09-12).

[158] D. Turull, P. Sjödin, and R. Olsson, "Pktgen: Measuring performance on high speed networks," *Computer Communications*, vol. 82, 03 2016.

[159] J. Stecklina, "Shrinking the hypervisor one subsystem at a time: A userspace packet switch for virtual machines," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 189–200. [Online]. Available: https://os.inf.tu-dresden.de/papers_ps/2014-vee-switch.pdf (Accessed 2022-08-21).

[160] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li, "Deconstructing xen." in *NDSS*, 2017. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_02A-4_Shi_paper.pdf (Accessed 2022-08-21).