SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Hyper-scalability of Network Interface Cards for Virtual Machines

Florian Dominik Freudiger

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Hyper-scalability of Network Interface Cards for Virtual Machines

Hyper-Skalierbarkeit von Netzwerkkarten für virtuelle Machinen

Author:	Florian Dominik Freudiger
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Peter Okelmann, M.Sc
Submission Date:	November 15, 2023

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

Munich, November 15, 2023 Florian Dominik Freudiger

Acknowledgments

I want to thank Peter Okelmann, my advisor, for their continued support throughout my thesis. The weekly meetings kept me on track and helped guide me throughout my thesis.

I also want to thank Prof. Dr.-Ing. Pramod Bhatotia and the entire TUM Chair of Computer Systems for allowing me to work on such an interesting topic accompanied by such wonderful people. I truly learned a lot.

Most importantly, I want to thank my family for always being by my side throughout my journey here at TUM.

Abstract

Emulation of network interface controllers (NICs) for use in virtual machines (VMs) is typically done in the hypervisor running the VMs. While this is fine for just running the emulated NICs, it is not ideal for building more complex applications requiring access to the NIC code. We introduce nic-emu, a library containing a behavioral model of the commonly used Intel E1000 NIC and more importantly, nic-emu-cli: a command line tool enabling its usage with the Quick Emulator (QEMU) hypervisor.

The project is open source and can be found at `https://github.com/vmuxIO/nic-emu`.

Contents

1 Introduction

Just like in physical machines, VMs require NICs to connect to networks. Traditionally one of two methods is being used to allow a VMs to access a NIC: passthrough or emulation, each of which has its own sets of limitations, mainly being that emulation tends to be slower than passthrough, while NIC passthrough cannot scale beyond a certain point since there is a limit to the number of VMs supported at once. In environments where thousands of VMs may run on a single host [23f], which all may need high-speed network access, only a fraction of VMs would be able to use the faster NIC passthrough, while the rest would be stuck with slower emulated NICs.

A dynamic solution is needed to overcome this where VMs can migrate between emulated and passed through NICs. Implementing such a complex system directly within the hypervisor code would be very complex. Instead, we aim to allow accomplishing such a system in userspace, outside the hypervisor. This thesis will test the viability of moving NIC emulation logic from within the hypervisor into a separate userspace process. To evaluate this, we implement a behavioral model of the Intel E1000 NIC, which we then integrate into a hypervisor using the libvfio-user framework. To evaluate this domain change, we will compare bandwidth and latencies to existing NIC emulators of QEMU.

2 Background

This chapter introduces the fundamental concepts and terms required to understand upcoming chapters.

2.1 Virtual machines

A system VM is in layperson's terms, a computer that is running within another computer. Instead of relying on physical hardware, a system VM is running on top of a hypervisor, which may use a combination of emulation, passthrough, and hardware acceleration techniques to provide a VM the same or similar foundation to be able to run as its physical counterpart.

There is another type of virtualization called process virtualization, used to provide platform-independent environments for applications to run in, such as the Java virtual machine. Nevertheless, as part of this thesis, all future references to virtualization and VMs will strictly be about system virtualization, not process virtualization. [23u]

2.2 Network Interface Cards

A network interface card or NIC is an essential computer system component for establishing network connections. NICs are incorporated within a system using a computer bus. [Mil23]

2.3 PCI

The Peripheral Component Interconnect (PCI) is a bus used to connect hardware to a computer. Using a standardized specification, various hardware components can be connected, from graphics accelerators to the aforementioned network interface cards.

A system can communicate with a connected PCI device using assigned address spaces for configuration, memory, and I/O access. [10]

On the other hand, a PCI device can also read and write data to the system's main memory using direct memory access (DMA).

2.3.1 Interrupts

For the device to alert the system to an asynchronous event such as network packet reception, interrupts are typically used. Interrupts allow the CPU to process other tasks instead of periodically polling the device for updates.

The CPU will then perform a context switch to the operating system's interrupt handler, triggering the corresponding subroutine in the device driver. [The23b] [23h]

The context switch and execution of an interrupt handler can, however, quickly pose a bottleneck when a connected device generates many events, as could be the case of high bandwidth packet reception on a modern NIC. This phenomenon is called an interrupt storm. [Kob20]

While operating systems can alleviate interrupt storms by disabling the interrupt [Kob20], a less drastic mitigation can be employed by the device using interrupt coalescing. That is enforcing an upper limit of interrupts sent in a given timeframe or through delaying interrupts under the assumption that more events might be triggered during the delay, so only one interrupt needs to be sent. [09] The drawback of these mechanisms is increased latencies between event generation and interrupt handler execution. [23g]

2.3.2 Single Root I/O Virtualization

Single Root I/O Virtualization (SR-IOV) is an extension to the Peripheral Component Interconnect Express (PCIe) standard, allowing a device to share its functionality between one or multiple VMs. To achieve this, the device can define multiple PCI functions of two types. Physical functions (PFs) are full devices that appear separate from one another. An example of having multiple PF can be found on some multi-port NICs, where each of its ports will be exposed as a separate PF. Virtual functions (VFs) are more simple PCI functions that may only have limited control over the device functionality. VFs derive from PFs, that is, a single PF can map many VFs.VFs may share a subset of the resources a PF controls. [17] [22]

In the case of NICs VFs, they may expose an interface for independently sending and receiving traffic without exposing critical functionality such as the ability to trigger PCI resets. [21]

These limitations make VFs perfect for sharing limited control with potentially untrusted systems such as third-party VMs, although care needs to be taken depending on the control shared. [15b]

3 Analysis

This chapter will provide an overview of the existing standard methods of exposing NICs to virtual machines and describe their advantages and disadvantages.

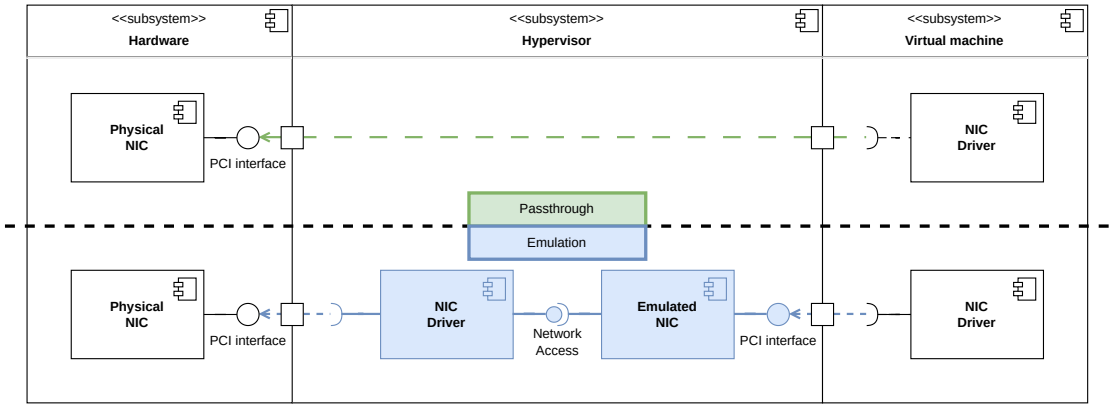

Figure 3.1: Component diagram of example passthrough and emulation deployments, differences colored.
When using emulation several more components and interface are required.

3.1 NIC Passthrough

As introduced in the last chapter, devices connected over PCI devices, including NICs, can be passed through to a virtual machine.

The VM can access the full performance of the device, but the device is stuck being attached to the VM until the VM operation concludes.

Passthrough has the immediate benefit of providing close to native performance, with the obvious drawback being that once a device has been passed through to a VM, it will be unavailable to the rest of the system. [Jon09]

Techniques such as SR-IOV do help by allowing NICs to provide many VFs, each of which can then be passed through to a different VM.

However, while SR-IOV in theory allows for up to 64000 VFs per PF, current modern high-speed NICs support a lot less. [12] For reference, NICs of the NVIDIA ConnectX®

family support up to 127 VFs per port. [23p] Similarly, NICs of the Intel Ethernet 810 Series supports a maximum of 256 VFs in total, spread evenly across up to 4 ports. [21]

While this might be sufficient for most applications, some scenarios may call for thousands of VMs per host. [23f]

Once all VFs have been assigned, subsequently created VMs might only be able to use NIC emulation, which we will analyze in the next section.

3.2 NIC emulation

When passthrough is not available or desired, emulation can be used to provide a VM with a NIC. Hypervisors such as QEMU can emulate a wide variety of common NICs, allowing both old and modern guest operating systems to be used. [23r]

Emulation does come with an added performance overhead since the behavior of the chosen NIC needs to be implemented in software and the bus used to connect it.

In order to optimize NIC emulation, paravirtualization can be used. QEMU supports the NIC of VirtIO, a standard defining a family of paravirtualized devices tuned towards efficient emulation. [Spe22] [Jon10] [The23c]

3.3 Missing balance

While paravirtualization helps, there may still be a large gap between passthrough and emulation. [Lin15]

The problem we are facing is that some environments, like serverless computing solutions such as Firecracker, potentially run thousands of VMs on the same host machine, an order of magnitude more than there are VFs available on modern NICs. [23f] [23p] [21]

In scenarios like these, only a fraction of VMs can be connected through VFs, this means many VMs are potentially stuck using slower emulated NICs. In contrast, the chosen VMs using passthrough might not fully utilize the available bandwidth.

Therefore, a dynamic solution is needed, where VMs can migrate between emulation and passthrough.

3.3.1 Traditional live migration

A similar problem occurs in the operation of data centers. Cloud providers of virtual computing resources must maintain the host machines running VMs. Live migration of VMs allows this without interrupting potentially critical customer processes. [23m] Live migration refers to migrating VMs between hosts without interruption by transferring

processor, memory, and connected device states. [Cla+05] Connected devices include attached NICs, so the first step has been analyzing if this could be applied to our problem, especially since we only need to migrate NICs, not the entire computing stack.

However, it is quite challenging migrating passed through VFs, especially between passthrough and emulation. Existing solutions typically require support and configuration of the guest operating system. In VMs running Linux, this is most commonly achieved by configuring bonding devices to switch between the connected devices automatically. PCI-Hotplug is then used to disconnect and reconnect devices. [Izu15] [23o] [Jon09]

This solution is far from ideal. Restricting guest operating system selection and requiring specific configuration is unsuitable for many environments, for example in cloud environments, where customers may provide custom VM images.

4 Design

As explained in the previous chapter, the problem of hyper-scaling VMs is that an imbalance arises when scaling past passthrough limits. In order to overcome this problem, we pursue introducing a mediating layer between VM and the attached NIC. In this chapter, we will therefore introduce the approach taken and the system components we implemented to test the viability of this approach.

4.1 Userspace emulation

Unfortunately, inserting such a layer will be difficult since the NIC emulation logic in hypervisors like QEMU resides deeply rooted in its code. [23q] Decoupling the existing NIC emulation logic in QEMU from its usages and adding this functionality in a codebase as large as QEMU will presumably be very difficult. Furthermore, this would only apply to QEMU, while some cloud providers may use different virtualization solutions. [23b]

To solve this complication, we pursued the mediation layer in userspace and provided separate emulation of a NIC. This way, our logic can run independently from the virtualization technology used.

4.2 System components

4.2.1 Behavioral model

For the NIC to emulate, we needed to choose a NIC that is also available in hardware for passthrough, effectively ruling out any para-virtualized devices such as the NIC within VirtIO.

We chose a NICs within the Intel 8254x family, commonly collectively referred to as Intel E1000, the driver's name. [23l]

NIC models within the family are not too complicated to emulate, given that various hypervisors implement them, including QEMU, which emulates three models of the NIC family. [23q] [15a]

Specifically, we chose the Intel PRO/1000 MT Desktop (82540EM), which is one of the models emulated by QEMU and also serves as its default NICs. [QEM23]

4.2.2 Hypervisor integration

Our userspace emulation will run in a process separate from the hypervisor. Because we still rely on an existing hypervisor, we need to integrate our emulated NIC into the chosen hypervisor. For example, this integration could be performed by accessing the same PCI bus that the hypervisor uses for its own included emulators. We would also need to select an inter-process communication (IPC) mechanism to communicate between the two processes.

To solve both issues of the hypervisor component and IPC, we utilized the libvfio-user framework. It is a framework for implementing PCI devices in userspace.

Once the desired PCI device parameters are configured, it hosts a UNIX socket over which clients can connect to issue PCI commands. As for hypervisors, cloud-hypervisor and QEMU support the framework as its clients, albeit support from QEMU is currently marked as "in-development" and is only available through building it from the source, from a set of patched "vfio-user" branches. [23k] [QN22]

4.2.3 Hypervisor

The hypervisor we use for testing our approach is QEMU. QEMU contains several of its own device emulators, which will serve as a reference and comparison to the behavioral model we use.

Since cloud-hypervisor is also supported by libvfio-user, integrating nic-emu into cloud-hypervisor should also work, although this has yet to be tested.

5 Implementation

In this chapter, we will dive deeper, explaining low-level implementation details.

5.1 Rust

We selected the Rust programming language to implement our standalone NIC emulator. [23s] Rust libraries offer powerful abstraction over upcoming components like the Linux TAP interface and the epoll system call. [dev23] [GN23]

5.1.1 libvfio-user-rs

The libvfio-user framework used has been written in the C programming language. Through the provided header files, it can easily be linked in C and C++ projects. [23k]

However, when using a different programming language such as Rust, the added intricacies of dealing with C types cause development to be quite cumbersome.

To solve this issue, we introduce libvfio-user-rs, a rust library abstracting much of the complexity of libvfio-user behind a more straightforward pattern-like structure.

```
vfu_ctx = vfu_create_ctx(VFU_TRANS_SOCK,
                         "/tmp/libvfio-user.sock",
                         LIBVFIO_USER_FLAG_ATTACH_NB,
                         &device,
                         VFU_DEV_TYPE_PCI);
if (vfu_ctx == NULL) {
    err(EXIT_FAILURE, "failed to create libvfio-user context");
}

int ret;
ret = vfu_pci_init(vfu_ctx, VFU_PCI_TYPE_CONVENTIONAL,
                   PCI_HEADER_TYPE_NORMAL, 0);
if (ret < 0) {
    err(EXIT_FAILURE, "vfu_pci_init() failed") ;
}
```

```
16
17 vfu_pci_set_id(vfu_ctx, 0x8086, 0x100e, 0x0000, 0x0000);
18 vfu_pci_set_class(vfu_ctx, 0x02, 0x00, 0x00);
19
20 ret = vfu_realize_ctx(vfu_ctx);
21 if (ret < 0) {
22     err(EXIT_FAILURE, "failed␣to␣realize␣device");
23 }
```

Listing 5.1: Example device initialization of the server.c libvfio-user sample with adjusted values [23k]

```
1  let config = DeviceConfigurator::default()
2      .socket_path(PathBuf::from("/tmp/libvfio-user.sock"))
3      .pci_type(PciType::Pci)
4      .pci_config(PciConfig {
5          vendor_id: 0x8086,
6          device_id: 0x100e,
7          subsystem_vendor_id: 0x0000,
8          subsystem_id: 0x0000,
9          class_code_base: 0x02,
10         class_code_subclass: 0x00,
11         class_code_programming_interface: 0x00,
12         revision_id: 0,
13     })
14     .non_blocking(true)
15     .build()
16     .unwrap();
17
18 // Produce a device of type Device using the defined config values
19 let mut device = config.produce::<Device>().unwrap();
```

Listing 5.2: Equivalent code excerpt in libvfio-user-rs, leveraging the builder pattern.

5.2 Behavioral model

As previously mentioned, the Intel 82540EM family of NICs is not too complicated to emulate. This is because of the simple interface over which the driver interacts with the NIC. All driver to NIC communication consists of accessing one of two PCI regions,

the memory-mapped I/O region and the port-mapped I/O region. Both regions are used to access the same memory. This memory consists of 32-bit registers that reflect and or control the state of the NIC. These 32-bit registers consist of various structures; they may contain booleans encoded as single bits set to 1 or 0 or can contain integers within them. For example, to trigger a device reset, the driver will simply set bit 26 of the control register, the first device register found at offset 0x0 to 1. The exact meaning of each bit of each register can be found in the device manual. [09]

Of the about 130 32-bit registers accessible on a physical NIC of this family of NICs, we emulate only 23. Many registers excluded from the emulation, such as statistical and diagnostic registers, are not essential to the NIC operation. Other functionality, like registers providing VLAN control, should be present for use in production environments but are not required for our limited testing.

As for NIC to driver communication, the NIC can use DMA to read and write from and into the system memory, which is being used to access one of the two descriptor rings.

A descriptor ring is a circular descriptor buffer allocated by the driver wherein the containing descriptors are structures containing information about received or to-be-transmitted Ethernet frames.

The NIC can also trigger interrupts to alert the driver of events such as the reception of a packet. As mentioned in the Background chapter, Interrupts can harm performance if they are triggered excessively, a phenomenon called interrupt storm. To avoid this, the emulated model also implements interrupt mitigation, where interrupts are delayed if time since the previous interrupt falls below a set threshold.

5.3 Libnic-emu

To allow other programs to integrate our behavioral model, we encapsulated it inside a library called "libnic-emu". It can then be linked to programs via Rust crate or static library file.

In order to integrate the library, the program will need to allow libnic-emu to:

- Send and receive ethernet frames
- Start, stop or pause packet reception to avoid packet overruns
- Use DMA to access descriptor rings
- Set or reset a timer required for interrupt mitigation, but this can be omitted if interrupt mitigation is not needed.

After that, the behavioral model can be used by accessing the PCI regions over its provided methods.

5.4 Nic-emu-cli

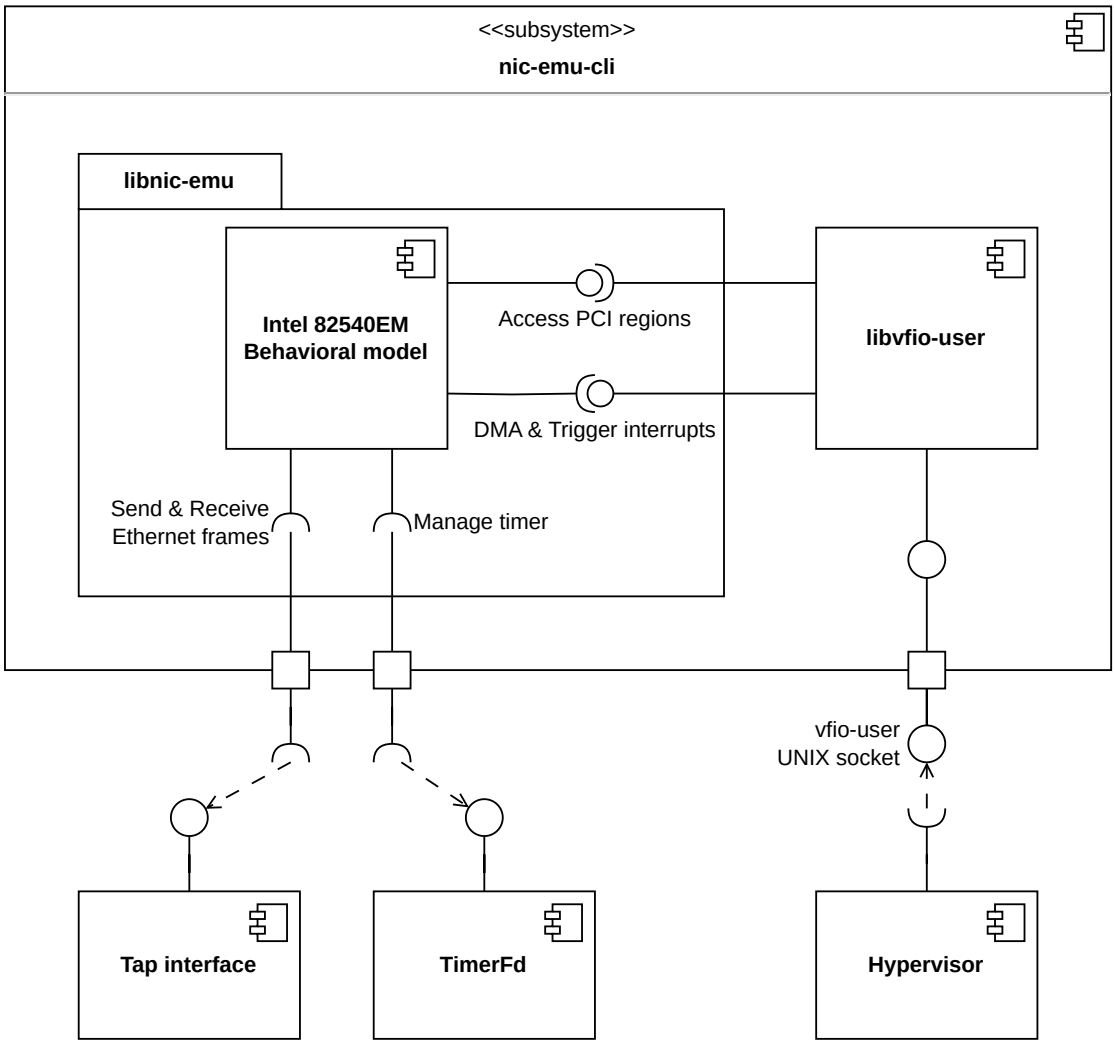

Figure 5.1: Diagram of the nic-emu-cli internal components and its external interfaces.

Nic-emu-cli combines the abovementioned libnic-emu and libvfio-user.

5.4.1 External components

In order to test libnic-emu, several other components are required. Since this setup is meant to be run on host systems running the Linux operating system, some of the following components will be Linux-specific. Nevertheless, similar components can also be found in other operating systems.

Tap interface

For the emulated NIC to receive and send packets, we use a TAP network interface, a virtual interface one can read and write to and from to send and receive Ethernet frames. [KYT02]

Once created, the TAP interface can be bound to a network bridge, another virtual network device meant to connect network interfaces. [The23a] The bridge could then be bound to a physical interface to send and receive packets like a passed-through NIC.

TimerFd

Libnic-emu also requires timers to implement interrupt mitigation. Linux's timerfd system calls provide an interface to operate a timer and receive events via a file descriptor.

5.4.2 Event loop

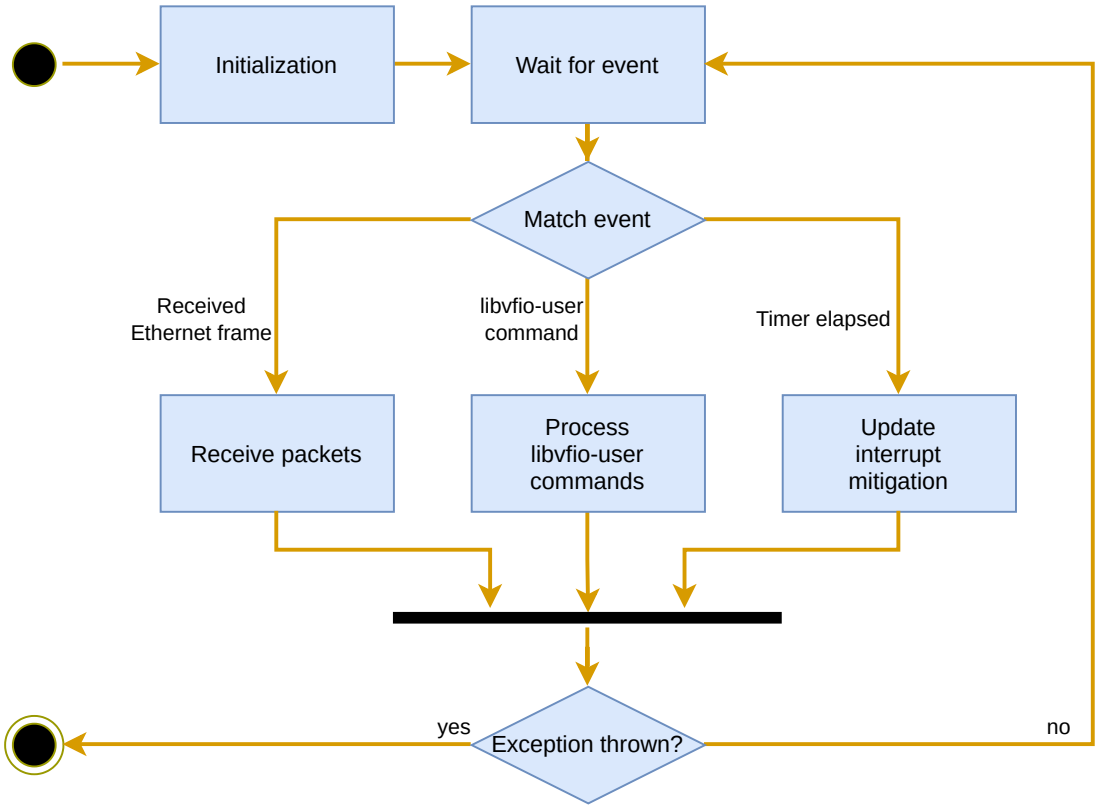

Figure 5.2: Current event loop of nic-emu-cli shown as an activity diagram. Events are all processed one after another, regardless of type.

We must process incoming ethernet frames from the tap interface, PCI commands received by libvfio-user, and elapsed timer events from timerfd. Since these events may occur in any order, we cannot wait and block for just one of them. Since all three components provide a file descriptor for event handling, we can use existing I/O multiplexing methods to wait for all three simultaneously.

6 Evaluation

This chapter will evaluate our userspace emulation against built-in emulators inside QEMU.

6.1 Research Questions

We will compare the emulator against the built-in E1000 NIC of QEMU. While not directly comparable because of the different codebase and programming language, this may yield a clue about the overhead caused by separating the emulation logic from QEMU into a process using libvfio-user for integration.

We will also compare against the VirtIO NIC of QEMU. It is expected that VirtIO should outperform both E1000 NIC emulators, given that it is built with emulation performance in mind. [Spe22]

That being said, as previously explained we need to emulate the same model as our potential passthrough device.

Instead, this will test whether this hybrid approach is viable at all, if the alternative is only using VirtIO.

To summarize, our research questions are as follows:

1. How big is the impact of moving device emulation into userspace?
2. Is using an emulated E1000 NIC instead of a VirtIO NIC viable?

6.2 Testbed

Our tests will be executed in 2 different configurations. The configurations differ from each other in the way our load generators and their counterparts are deployed and bound to the virtual machine. The specific details will be listed within each of the tests.

To run our virtual machine, we are using QEMU version 7.1.50 with the applied libvfio-user patches, which can be found in the "vfio-user-7.1.5" QEMU branch on GitHub. [QN22]

Nic-emu-cli is being built in its release profile using cargo and rustc at version 1.70.0. In addition to the default release profile optimizations, link-time optimization has been

enabled, panic behavior set to terminate on panic, and the number of code generation units has been set to one to avoid missing optimizations. [23c]

The virtual machine is hosted on a Linux server using a 12-core, 24-thread Intel Xeon Gold 5317 with a base processor frequency of 3.00 GHz. To minimize test variance, the Intel turbo boost technology has been disabled. The server uses 256GB of buffered DDR4 memory at 2933 MT/s and is running NixOS 23.05 (Stoat) using Linux Kernel version 5.15.84. [23d]

The virtual machine itself has been assigned one vCPU and 16GiB of RAM. It is running NixOS 23.11 (Tapir) using Linux Kernel version 6.1.38.

All tests use a virtual TAP interface. QEMU NIC emulation is tested by opening the TAP interface using the "-netdev" command-line argument and assigned to a "virtio-net-pci" or "e1000" device, depending on what is being tested. For testing userspace emulation, nic-emu-cli is being started before QEMU. Nic-emu-cli will open the particular TAP interface using the "–tap <TAP>" argument and will host a libvfio-user socket, which will then be bound as a "vfio-user-pci" device in QEMU.

6.3 Bandwidth comparison

For our first test, we want to evaluate the maximum bandwidth the emulated NICs can sustain. To measure this, we will use iPerf. iPerf is a measurement tool that evaluates the maximum bandwidth on IP networks. We will use iPerf3, the latest release of iPerf. [23j] [23i]

iPerf3 works by first hosting a server on a target machine and a client on the source machine. The client will then try to send as much data as possible to the server in a given timeframe. The resulting amount of data the server receives will then be used to calculate the average bandwidth the connection sustained.

6.3.1 Test Setup

Since iPerf3 operates on the transport layer using either TCP or UDP, we assign both the TAP interface and the ethernet interface within the VM unique IPv4 addresses within the same subnet.

The iPerf3 server will be run inside the VM, after which the client will be started on the same server running the VM.

We will run iPerf3 in its default TCP mode for 30 seconds. We set iPerf3 to omit the first second of the test from the result to skip the period during which the TCP slow start mechanism builds up the maximum size of its congestion window. [23t]

We will measure both the average bitrate sending data from the client to the server (RX) and the reverse (TX).

6.3.2 Results

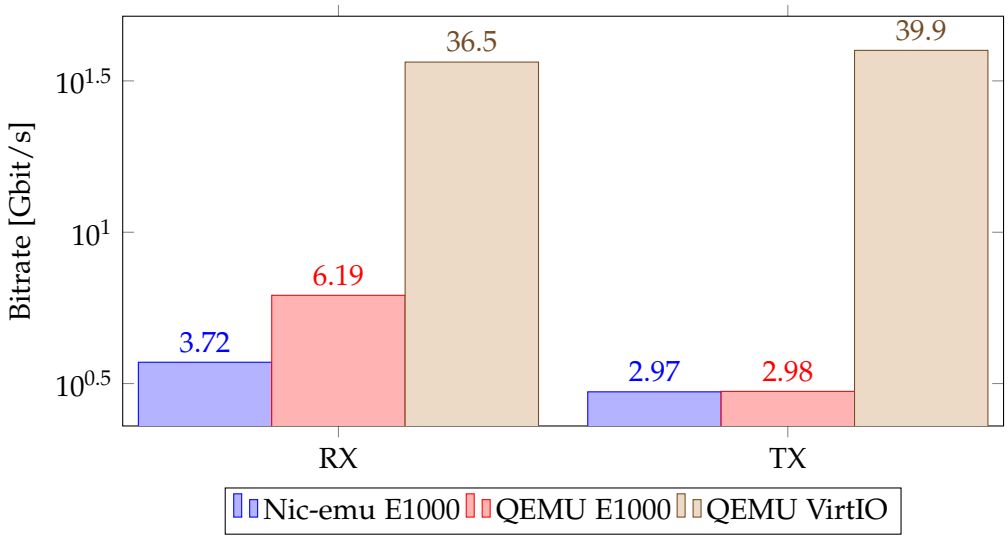

Figure 6.1: Logarithmic bar chart of iPerf3 test results.

E1000 vs E1000

The differences between the emulated NICs are quite large. Comparing our emulated E1000 NIC to the one inside QEMU, we can see the RX bandwidth is 66% higher using the E1000 NIC in QEMU.

Interestingly, the TX bandwidth between the two is nearly identical. Pinpointing the exact cause of this significant performance difference, which only affects RX but not TX bandwidth, is not trivial because of the completely different codebases. However, we can speculate possible causes:

- QEMU may use a major optimization for E1000 NIC RX. This would also explain the performance dissimilarity between RX and TX in the QEMU E1000.
- The E1000 NIC of QEMU could naturally perform better than its counterpart in nic-emu. However, a performance bottleneck such as TCP segmentation offloading limits the TX performance of both.

E1000 vs VirtIO

Comparing both E1000 NICs to the VirtIO NIC, we can see VirtIO outperforms both E1000 NICs by a factor of 6-13x. This difference can be attributed to the design of

VirtIO, which has been designed with emulation performance in mind. [Spe22; Jon10; The23c]

6.4 Latency comparison

To measure latencies, we will be using Moongen. Moongen is a high-speed packet generator that is controlled using a Lua script. In our case, we will use it to measure the latencies involved in sending ethernet frames through emulated NICs.

Unlike iPerf3, Moongen sends ethernet frames without any transport protocol such as TCP or UDP. It can scale to saturating high-speed links because it has been built using the Data Plane Development Kit (DPDK) [Emm23], a set of libraries that allows applications running in Linux userland to send and receive packets from a specific NIC in a high throughput fashion. [23n]

6.4.1 Test Setup

Because Moongen is directly using a DPDK compatible NIC, the setup for these measurements will be different, running the load generation on a separate host.

The second host will be a server using the same CPU and memory combination as our VM host previously discussed. It also uses the same operating system but with a newer Linux kernel version, 6.5.10. [23e]

Both hosts are connected through an Intel E810 NICs, capable of 100Gbit/s. The Intel E810 NICs will be bound to DPDKs on the load generation server. On the VMs host server, the network interface corresponding to its Intel E810 will relay its packets to the TAP interface using a bridge device.

Unlike iPerf3, we will not have a server component acknowledging the received packets. Instead, we use "xdp-reflector", a simple eXpress Data Path (XDP) program that will reflect incoming ethernet frames to its sender.

This setup has the added benefit of reflecting latency in a much more realistic manner since the network traffic of a dynamic NIC mediation solution would also pass physical NICs.

6.4.2 Results

Low load

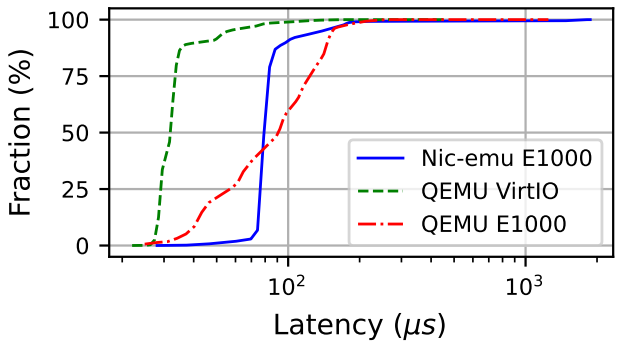

Figure 6.2: Comparison of latencies at low load using cumulative distribution function

First, we will evaluate a low-load scenario, during which packets arrive more spaced out from one another, which should cause interrupt mitigation timers to be hit less often.

To measure this, we configure Moongen to send 60B packets at a rate of 10kpps.

In the graph, we can see that both emulated QEMU NICs have much lower minimum latencies than the nic-emu E1000 NIC. This result is expected since sending and receiving packets over nic-emu has the added latency of the vfio-user communication between the libvfio-user framework and QEMU.

Despite everything, both E1000 NIC emulators have similar average latencies, with the nic-emu NIC having a lower latency than the E1000 QEMU NIC around half the time. One possible explanation for this is a difference in the way interrupt mitigation is accomplished in the QEMU E1000 NIC.

There are several ways to mitigate interrupts in the physical E1000, controlled by different registers. While the QEMU E1000 emulates three such interrupt mitigation registers, nic-emu only emulates one. In addition to the added registers, QEMU E1000 also limits interrupts to a maximum of 7813 interrupts per second, since this number was mentioned as an upper bound in the NIC manual. [09] [23q] Nic-emu has not implemented such an upper bound. Because of the stricter interrupt mitigation, we can assume more interrupts are being mitigated, which could cause the increased latency measured. This assumption is further substantiated by the QEMU E1000 curve being similar in both low-load and high-load graphs. Therefore, interrupt mitigation

is presumably activated prematurely in the QEMU E1000 NIC, as we can see in the behavior of the VirtIO NIC, only mitigating during high loads.

If the QEMU E1000 would not employ such strict interrupt mitigation, the added latency of vfio-user would likely also be visible in the average and maximum latency difference.

However, while E1000 NIC latency is similar, VirtIO NIC exhibits much lower latencies, presumably because of its emulation-oriented design.

High load

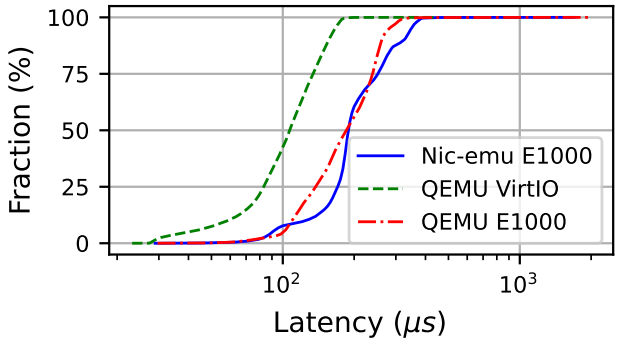

Figure 6.3: Comparison of latencies at high load using cumulative distribution function

For our high-load test, we will use the same Moongen configuration as our low-load test with a ten times higher packet rate, sending 60B packets at a rate of 100kpps.

Here, we can clearly see the effect of interrupt-mitigation in work. All NICs actively mitigate interrupts and generate similar curves. Each NIC is delaying packet events up to a certain threshold, stretching a near-linear curve between minimum and maximum latency.

6.5 Evaluating research questions

The first research question was regarding how significant the impact of moving device emulation into userspace is.

We saw that the main difference between the E1000 NICs is the increased minimum latency likely caused by the additional vfio-user communication. The QEMU E1000 NIC could also sustain a higher packet reception bandwidth, although this may be

caused by missing optimizations in nic-emu. Therefore, moving device emulation into userspace comes with the main drawback of potentially higher latencies, that is, if emulation in both hypervisor and userspace employs the same interrupt mitigation strategy.

Secondly, we wanted to know whether an emulated E1000 NIC is viable instead of using VirtIO emulation. Considering VirtIO could achieve much higher maximum bandwidths while maintaining lower latencies, we can confidently say that it is not.

Regardless of the E1000 NIC emulation results, one should also keep in mind that the E1000 family of physical NICs is only made to support 1Gbit/s ethernet links, slower than both E1000 emulators. [09]

Because of this, the dynamic NIC emulation-passthrough model previously mentioned would not be beneficial, given that all E1000 NICs, including emulated and physical models, are much slower than the emulated VirtIO NIC.

7 Related Work

In this chapter, we will explore one related work concerned with the performance of emulated NICs within QEMU.

7.1 Speeding Up Packet I/O in Virtual Machines [RLM13]

This paper explores avenues to enable faster network performance of existing emulated NICs like the Intel E1000 for both guest-to-host and guest-to-guest networking within the QEMU hypervisor.

7.1.1 Optimizations

The first problem discovered in the existing E1000 NIC emulation is the incomplete interrupt mitigation strategy. To address this, the work emulates additional interrupt mitigation registers in QEMU.

Additionally, tweaks to the E1000 Linux driver configuration, such as "Send Combining" are being used to further minimize the amount of interrupts generated.

One significant optimization performed is the addition of a "Communication Status Block or CSB" within the E1000 emulator and driver. This CSB is akin to shared memory regions of para-virtualized devices and is used to reduce the amount of VM exits.

Last, the VALE software switch has been integrated into the QEMU backend, improving network performance between multiple VMs. [RLM13]

7.1.2 Results

The optimizations lead to a major speed-up of the emulated E1000 NIC of QEMU, being close to or matching the performance of the VirtIO NIC, depending on the modifications such as the VALE backend used. [RLM13]

7.1.3 Comparison

As for similarities, both the paper and this thesis work closely with the Intel E1000 NIC. This thesis aims to test the viability of userspace NIC emulation to enable passthrough

integration and migration. This paper, on the other hand, is instead trying to optimize the NIC emulation through various techniques such as para-virtualization, which could be a way to extend the viability of emulation in high-performance requiring scenarios instead of having to resort to passthrough.

One should note that this paper pre-dates many of the latest changes of the emulated E1000 NIC in QEMU. Since then, the E1000 emulation code has received various changes and fixes, including additions to its interrupt mitigation logic. [23q]

8 Conclusion

In this thesis, we have implemented a userspace NIC emulator, an emulator running in a separate process from the hypervisor. It is capable of high-performance operation, behaving similarly to the existing E1000 NIC emulator of QEMU. We also introduced a Rust library wrapping the libvfio-user framework using common design patterns.

While we found that E1000 NIC emulation performance to be slow when compared against the state-of-the-art VirtIO emulated NIC, we did find that the impact of moving emulation into userspace is likely to be minor.

nic-emu
E1000 emulation library and application
`https://github.com/vmuxIO/nic-emu`

libvfio-user-rs
Rust wrapper of libvfio-user
`https://github.com/vmuxIO/libvfio-user-rs`

9 Future Work

9.1 Parallel event processing

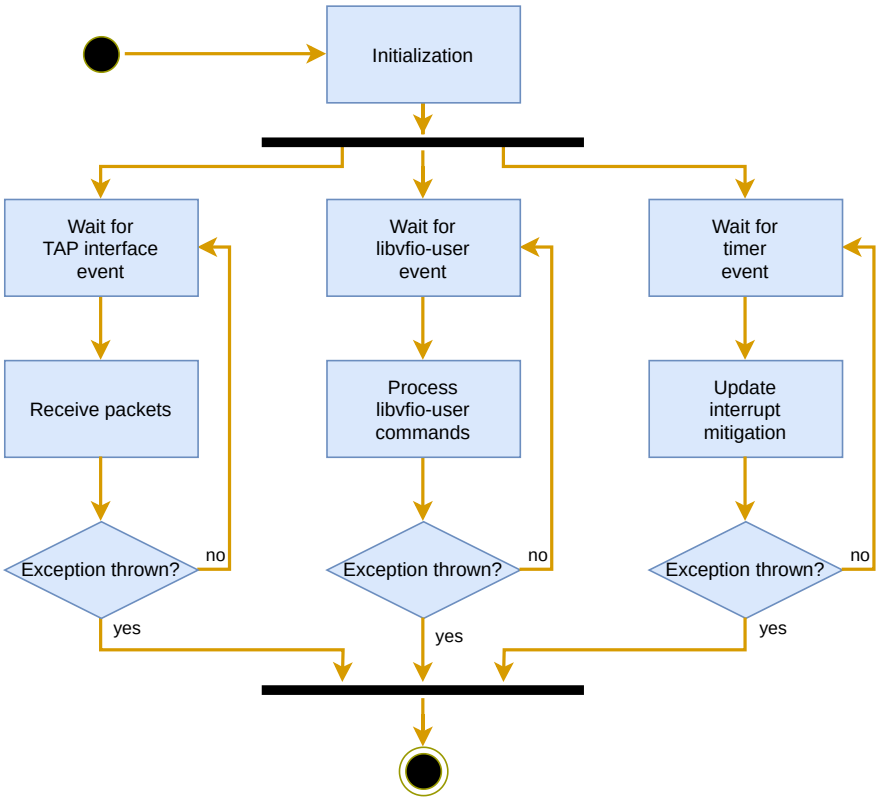

Figure 9.1: The optimized event loop of nic-emu-cli shown as an activity diagram. The different event types would be processed in parallel to each other.

Currently, nic-emu-cli processes events synchronously, meaning that events arriving during the processing of an unrelated event will be left waiting until the previous event has finished processing. This holdup could become a bottleneck during duplex operation since the emulated NIC cannot send and receive packets simultaneously.

To avoid this, the event loop could be parallelized via multi-threading. This change does however come at the cost of increased complexity, as nic-emu-cli would need to be redesigned to allow access to its components in a thread-safe manner since data races and deadlocks could be fatal, potentially corrupting packets at best, causing unrecoverable exceptions at worse.

9.2 Faster NIC

Evaluation shows that the E1000 NIC emulation performance lacks quite a bit behind the emulated VirtIO NIC. Emulating a behavioral model of another, more modern NIC such as a NIC of the Intel E810 series could be worthwhile, especially if one would implement the previously mentioned dynamic NIC mediation system, where passthrough would not be beneficial if the physical NIC is slower than its emulated counterpart. A physical Intel E810 NIC should generally outperform VirtIO, considering the VirtIO bandwidth measurements all measured below even half of its rated 100Gbit/s maximum bandwidth. [23a]

Abbreviations

NIC network interface controller

PCI Peripheral Component Interconnect

PCIe Peripheral Component Interconnect Express

PF physical function

VF virtual function

DMA direct memory access

VM virtual machine

QEMU Quick Emulator

SR-IOV Single Root I/O Virtualization

IPC inter-process communication

DPDK Data Plane Development Kit

XDP eXpress Data Path

List of Figures

Bibliography

[09] *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual*. Intel Corporation. 2009.

[10] *Writing Device Drivers - PCI Address Domain*. Oracle Corporation. 2010. URL: https://docs.oracle.com/cd/E19253-01/816-4854/hwovr-25/index.html (visited on 31/10/2023).

[12] *Oracle VM Server for SPARC 2.2 Administration Guide*. Oracle and/or its affiliates. 2012. URL: https://docs.oracle.com/cd/E35434_01/html/E23807/usingsriov.html (visited on 03/11/2023).

[15a] *Choosing a network adapter for your virtual machine (1001805)*. VMware, Inc. 2015. URL: https://kb.vmware.com/s/article/1001805 (visited on 14/11/2023).

[15b] 'Securing Self-Virtualizing Ethernet Devices'. In: Technion - Israel Institute of Technology. 2015.

[17] *Virtualization Host Configuration and Guest Installation Guide - Chapter 13. SR-IOV*. Red Hat, Inc. 2017. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_host_configuration_and_guest_installation_guide/chap-virtualization_host_configuration_and_guest_installation_guide-sr_iov (visited on 31/10/2023).

[21] *SR-IOV Configuration Guide - Chapter 13. SR-IOV*. Intel Corporation. 2021.

[22] *Network Driver Design Guide - Overview of Single Root I/O Virtualization (SR-IOV)*. Microsoft. 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov- (visited on 06/11/2023).

[23a] *100GbE Intel® Ethernet Network Adapter E810*. Intel Corporation. 2023. URL: https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters/e810-network-adapters/products.html (visited on 14/11/2023).

[23b] *AWS Nitro System*. Amazon Web Services. 2023. URL: https://aws.amazon.com/ec2/nitro/ (visited on 05/11/2023).

[23c] *Codegen Options*. The Rust Foundation. 2023. URL: https://doc.rust-lang.org/rustc/codegen-options/index.html (visited on 13/11/2023).

[23d] *doctor-cluster-config: river*. TUM - Chair of Computer Systems. 2023. URL: https://github.com/TUM-DSE/doctor-cluster-config/blob/26e84a48e3e9a8cd83354c30e2684f06b5356e38/docs/hosts/river.md (visited on 13/11/2023).

[23e] *doctor-cluster-config: wilfred*. TUM - Chair of Computer Systems. 2023. URL: https://github.com/TUM-DSE/doctor-cluster-config/blob/26e84a48e3e9a8cd83354c30e2684f06b5356e38/docs/hosts/wilfred.md (visited on 13/11/2023).

[23f] *Firecracker*. Amazon Web Services. 2023. URL: https://firecracker-microvm.github.io/ (visited on 04/11/2023).

[23g] *IBM AIX documentation - Interrupt coalescing*. IBM Corporation. 2023. URL: https://www.ibm.com/docs/en/aix/7.3?topic=options-interrupt-coalescing (visited on 01/11/2023).

[23h] *IBM AIX documentation - Interrupt handlers*. IBM Corporation. 2023. URL: https://www.ibm.com/docs/en/aix/7.3?topic=hierarchy-interrupt-handlers (visited on 31/10/2023).

[23i] *iperf2 / iperf3*. ESnet, Lawrence Berkeley National Laboratory. 2023. URL: https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf/ (visited on 06/11/2023).

[23j] *iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool*. ESnet, Lawrence Berkeley National Laboratory. 2023. URL: https://github.com/esnet/iperf (visited on 06/11/2023).

[23k] *libvfio-user*. Nutanix. 2023. URL: https://github.com/nutanix/libvfio-user (visited on 29/10/2023).

[23l] *Linux PRO/1000 Ethernet Driver*. Intel Corporation. 2023. URL: https://github.com/torvalds/linux/tree/master/drivers/net/ethernet/intel/e1000 (visited on 29/10/2023).

[23m] *Live migration process during maintenance events*. Google Cloud. 2023. URL: https://cloud.google.com/compute/docs/instances/live-migration-process (visited on 04/11/2023).

[23n] *MoonGen*. Canonical. 2023. URL: https://ubuntu.com/server/docs/network-dpdk (visited on 13/11/2023).

[23o] *NVIDIA MLNX_OFED Documentation Rev 5.4-1.0.3.0 - SR-IOV Live Migration*. NVIDIA. 2023. URL: https://docs.nvidia.com/networking/display/mlnxofedv541030/sr-iov+live+migration (visited on 04/11/2023).

[23p] *NVIDIA MLNX_OFED Documentation v5.8-3.0.7.0.101 for DGX H100 Systems - Single Root IO Virtualization (SR-IOV)*. NVIDIA. 2023. URL: https://docs.nvidia.com/networking/display/mlnxofedv583070101/single+root+io+virtualization+(sr-iov) (visited on 03/11/2023).

[23q] *QEMU e1000 emulation*. 2023. URL: https://github.com/qemu/qemu/blob/master/hw/net/e1000.c (visited on 29/10/2023).

[23r] *qemu/hw/net*. QEMU project developers. 2023. URL: https://github.com/qemu/qemu/tree/master/hw/net (visited on 04/11/2023).

[23s] *Rust*. The Rust Foundation. 2023. URL: https://www.rust-lang.org/ (visited on 05/11/2023).

[23t] *TCP slow start*. Mozilla Corporation. 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/TCP_slow_start (visited on 13/11/2023).

[23u] *What is a virtual machine?* VMware, Inc. 2023. URL: https://www.vmware.com/topics/glossary/content/virtual-machine.html (visited on 07/11/2023).

[Cla+05] C. Clark, S. H. Keir Fraser, J. G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield. 'Live Migration of Virtual Machines'. In: 2005.

[dev23] tokio-jsonrpc developers. *tuntap*. 2023. URL: https://github.com/vorner/tuntap (visited on 05/11/2023).

[Emm23] P. Emmerich. *MoonGen*. 2023. URL: https://github.com/emmericp/MoonGen (visited on 13/11/2023).

[GN23] S. Glavina and J. Nunley. *polling*. 2023. URL: https://github.com/smol-rs/polling (visited on 05/11/2023).

[Izu15] T. Izumi. *Live Migrate Guests w/PCI Pass-Through Devices*. 2015.

[Jon09] M. Jones. *Linux virtualization and PCI passthrough*. IBM Corporation. 2009. URL: https://developer.ibm.com/tutorials/l-pci-passthrough/ (visited on 03/11/2023).

[Jon10] M. Jones. *Virtio: An I/O virtualization framework for Linux*. IBM Corporation. 2010. URL: https://developer.ibm.com/articles/l-virtio/ (visited on 31/10/2023).

[Kob20] K. Kobayashi. *Development of 'Interrupt Storm Detection' Feature*. 2020.

[KYT02] M. Krasnyansky, M. Yevmenkin and F. Thiel. *Universal TUN/TAP device driver*. 2002. URL: https://www.kernel.org/doc/html/v5.8/networking/tuntap.html (visited on 04/11/2023).

[Lin15] Linux kvm wiki community. *10G NIC performance: VFIO vs virtio*. 2015. URL: https://www.linux-kvm.org/page/10G_NIC_performance:_VFIO_vs_virtio (visited on 14/11/2023).

[Mil23] S. Miller. *What Is a Network Interface Card (NIC)?* 2023. URL: https://www.codecademy.com/resources/blog/network-interface-card/ (visited on 29/10/2023).

[QEM23] QEMU project developers. *Documentation/Networking*. 2023. URL: https://wiki.qemu.org/Documentation/Networking (visited on 05/11/2023).

[QN22] QEMU project developers and Nutanix. *qemu - vfio-user-7.1.5*. 2022. URL: https://github.com/oracle/qemu/tree/vfio-user-7.1.5 (visited on 29/10/2023).

[RLM13] L. Rizzo, G. Lettieri and V. Maffione. 'Speeding up Packet I/O in Virtual Machines'. In: *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '13. San Jose, California, USA: IEEE Press, 2013, pp. 47–58. ISBN: 9781479916405.

[Spe22] O. C. Specification. *Virtual I/O Device (VIRTIO) Version 1.2*. 2022.

[The23a] The Archlinux wiki community. *Network bridge*. 2023. URL: https://wiki.archlinux.org/title/network_bridge (visited on 04/11/2023).

[The23b] The LDP community. *The Linux Kernel Module Programming Guide - Chapter 12. Interrupt Handlers*. 2023. URL: https://tldp.org/LDP/lkmpg/2.6/html/x1256.html (visited on 31/10/2023).

[The23c] The OSDev community. *Virtio*. 2023. URL: https://wiki.osdev.org/Virtio (visited on 31/10/2023).