Università di Pisa

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Computer Engineering

Tesi di Laurea Magistrale

High performance networking extensions for VirtualBox

Candidato:
Luca Carotenuto
Matricola 468020

Relatori:
Prof. Luigi Rizzo
Prof. Giuseppe Lettieri

Anno Accademico 2015-2016

Alla mia famiglia,
a Erika,
ai miei amici.

Abstract

Virtual Machine systems are commonly used in several organizations providing network services, since those systems supply high reliability, security and availability. Therefore, network performance has become a critical issue to deal with, since Virtual Machine systems are widespread nowadays.

In this thesis we are going to present VirtualBox hypervisor, giving some details about its architecture and analyzing network performances of the existing solution. We then implement an extension that interfaces the hypervisor with netmap framework [1], which provides fast packet I/O. Finally, we present some optimizations to an emulated network device (e1000 in our case), that considerably improve network performances.

Contents

Chapter 1

Introduction

It is important to point out that the term *Virtual Machine* may have multiple interpretations, so we must first specify which one of those meanings we are referring to.
When we talk about Virtual Machine (VM), we refer to a *virtualized* computing environment running on top of a physical computing environment; as a result, we get one or more independent VMs, which may be different from the original one.
Before proceeding, we introduce some terminology:

- *Guest*: The VM.
- *Host*: The physical computing environment that *hosts* one or more VMs.
- *Virtual Machine Monitor (VMM)*: The software part that provides support for virtualization. Also known as *Hypervisor*.

The main reason that caused the spread of VMs is the abstraction levels that it introduces; this brings many benefits:

- *Flexibility*: you can run programs compiled for a given Instruction Set Architecture (ISA) and/or a given Operating System (OS) on top of a computer that has a different ISA and/or different OS (e.g. you can test new software on different architectures without **having** one machine per architecture).
- *Protection*: each guest is **isolated**, which means that you can execute different applications on different VMs, so that if an application has a security

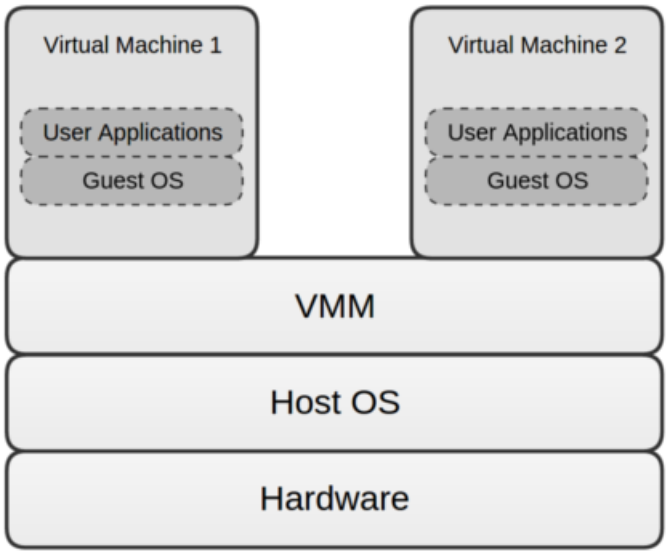

Figure 1.1: Type 2 System Virtual Machine generic architecture

issue, only the VM (or VMs) running that specific software will be exposed to it.

- *Resources usage*: one single physical machine may provide multiple services using the 100% of the resources, instead of using many underutilized physical machines, thus reducing costs and saving energy.

- *Mobility*: replicating VMs to other locations is only a matter of copying/-transmitting some files; this helps avoiding multiple setups since through a VM you can bring a functioning computing environment ready to use to the user.

As stated before, the term *Virtual Machine* may have several meanings. A generic architecture of the ”class” of VMs we will refer to (called *Type 2 System Virtual Machines*), is shown in figure 1.1. In this case the VMM is a regular OS process, that runs in the host OS along with other processes. The VMM can access the physical resources through the OS services, which depend on the specific OS. We will not investigate other classes of Virtual Machines since this topic falls outside this work.

1.1 Virtual Machine Implementation

The basic idea behind VMs, is to *emulate*, i.e. to execute code written for a certain environment, using another environment. In the following, we will briefly present the three basic techniques to implement emulation: interpretation, dynamic translation, hardware-based virtualization.

1.1.1 Interpretation

This is the naive emulation technique. The VMM has to perform in software what a physical CPU would have done in hardware: so it will be implemented as a loop, for each iteration, performs the fetch, decode and execute phases of instruction execution.

Writing an interpreter for a modern ISA can be a very long and difficult process, even if it is conceptually simple; in fact, it is just a matter of reading an Instruction Set specification and implement all the possible instructions respecting the specifications.
However, the simplicity of this approach is responsible for its inefficiency; as a matter of fact, for each source instruction, the VMM has to execute many host instructions (e.g. 30-100) to perform in software all the necessary operations. The average *translation ratio* is very high (e.g. 40).

1.1.2 Dynamic translation

This is a more sophisticated form of emulation. Rather than performing a ”source-code-to-source-code” translation, the idea is to translate it into an equivalent binary code that can be executed directly on the host CPU.
This method amortizes the cost of interpretation, doing the fetch and decode phases only once or a few times. The code execution step of an instruction or a block of instructions is generated once (or a few times) and stored in a *Code Cache*. After some time the code cache will contain the complete translation of the source program into the host ISA.
As a result, the average translation ratio can be close to 1, giving an acceptable

performance.

This technique is way more complicated than the previous one. In this case, several problems are present:

- *code-discovery*: makes impossible to do static translation
- *code-location*: different address space of the guest and host systems
- *state mapping*: the way the VMM maps guest registers and the like to the host ones

It is interesting to notice that both interpretation and dynamic translation can make sense also in the case that guest and host have the same ISA; if this is the case, the translation is simplified since the code can be natively executed on the host machine, without performance losses.
However there are some cases where emulation in software my be necessary. As a typical example, memory accesses to the I/O space may need software emulation. In particular, if the guest wants to access a physical resource that is present on the host (e.g. a network adapter), the VMM cannot allow direct access to the device, because other processes could be accessing the same device at the same time, and, obviously, the host network driver and the guest network driver are not aware of each other. On the other hand, if the guest wants to access a virtual device (which does not exist on the host), the I/O instruction must be *trapped*[1] in order to emulate the device behavior in software.

1.1.3 Hardware-based virtualization

Due to the widespread use of VMs, extensions for virtualization were introduced by processor vendors. Thanks to these hardware assists, some of the problems affecting dynamic translation techniques have been overcome, and at the same time they have made it easier to execute guest code natively. Both AMD and Intel proposed their extensions for the x86 ISA, AMD-v ([4]) and VT-x ([5]) respectively.

[1] Guest execution is interrupted and the VMM takes control.

With this new extension, the CPU can execute in two different modes: *root mode* and *VM mode* (or *non-root mode*). The CPU can switch from root mode to VM mode through a so called *VM entry* instruction, while can switch back to root mode through a so called *VM exit* instruction. When in VM mode, the CPU can execute guest code in a controlled environment, i.e. the CPU cannot execute some safety-critical instructions (e.g. I/O instructions); when necessary, CPU performs a VM exit and runs host code (VMM or other processes). The switch operation between host world and guest world is similar to a context switch, since it involves the saving of the host state and loading the guest state (and vice versa). Although performed in hardware, these transitions between host and guest worlds are expensive in terms of performance, because software overhead, OS operations and userspace/kernelspace transitions are involved in the switching operations, but they are also necessary when dealing with I/O operations or interrupts. Hence, VM switches must be minimized in order to achieve good I/O performances.

1.2 I/O Virtualization techniques

Emulating a device means doing in software what the device would do in hardware. Thus, when a guest accesses an I/O device (e.g. writes to a device register), the VMM must take over and emulate all the operations associated with the specific I/O access.

In order to improve I/O virtualization techniques, three approaches have been defined:

- Hardware support in the devices (*virtual functions* and IOMMU [6]), so that a guest can directly access devices in a protected way and run at native speed.
- Runtime optimizations in VMM. E.g. running short code involving multiple I/O instructions in interpreted mode saves some VM exits[2].

[2]See [7] for details.

- Design *virtual* device models in order to reduce expensive operations in device emulations (e.g. I/O accesses and interrupts). This approach is known as *device paravirtualization* and produced some virtual device models, such as VirtIO ([8]). This requires synchronization and memory sharing between the guest and VMM in order to exchange information, while interrupts are used only for notification purposes. In that way it is easier to minimize the amount of VM exits.

Chapter 2

VirtualBox

In this chapter we will present VirtualBox hypervisor, giving details about its features (section 2.1), its internal architecture (section 2.2), and how the behavior of e1000 device is emulated (section 2.4).
As host OS, we used Ubuntu 15.10 64-bit with kernel version 4.2.0. The guest OS is the same as the host one.

2.1 VirtualBox features

VirtualBox is a free, open source hypervisor, written entirely in C/C++. In particular, it is a cross-platform type 2 VMM, so it is able to run an arbitrary OS, regardless of the host OS, and it is implemented as a regular process in the host OS, therefore it can make use of all OS services. At the time of the writing, VirtualBox version number is 5.0.4, so we will refer to that version for Linux OS (since our host OS is Linux based).

Here is a brief outline of VirtualBox main features:

- **Portability.** VirtualBox runs on a large number of a 32-bit and 64-bit host operating systems. It can run VMs created on different hosts and/or with different virtualization software.
- **Multiple virtualization interfaces.** VirtualBox provides three different virtualization interfaces

  - *Minimal*: Announces the presence of a virtualized environment.
  - *KVM*: Presents a Linux KVM hypervisor interface which is recognized by Linux kernels starting with version 2.6.25.
  - *Hyper-V*: Presents a Microsoft Hyper-V hypervisor interface which is recognized by Windows 7 and newer operating systems.

  We chose the KVM interface for tests and implementations.

- **Multiple frontends**: A *frontend* is a user interface that VirtualBox provides, such as:
  - *VBoxManage*: A textual interface that allows advanced settings for VMs.
  - *VirtualBox*: The default frontend, based on Qt [11].
  - *VBoxSDL*: An alternative frontend based on SDL [12]. This is useful for business use as well as testing during development. The VMs then have to be controlled with VBoxManage.
  - *VBoxFB*: The "Framebuffer GUI", a GUI that sits directly on the Linux framebuffer. Not currently maintained.

- **No hardware virtualization required.** Even if hardware virtualization is fully supported, VirtualBox does not require the processor features such as Intel VT-x or AMD-V; in that way VirtualBox can be used also on old hardware which has not these features.

- **Guest Additions.** VirtualBox Guest Additions are software package which can be installed *inside* of supported guest systems to provide additional integration and communication with the host system (e.g. accelerated 3D graphics, automatic adjustment of video resolution and more).

- **Great hardware support.** Among others, VirtualBox supports:
  - **Guest multiprocessing (SMP).** VirtualBox can present up to 32 virtual CPUs to each virtual machine, regardless of how many CPU cores are physically present on the host.

  - **USB device support**. VirtualBox implements a virtual USB controller that allows to connect arbitrary USB devices to VMs without having to install device-specific drivers on the host.
  - **Hardware compatibility.** VirtualBox virtualizes a vast array of virtual devices. That includes IDE, SCSI and SATA hard disk controllers, several virtual network cards (including e1000) and so on.
  - **Full ACPI support.** The Advanced Configuration and Power Interface (ACPI) is fully supported by VirtualBox.

- **Multigeneration branched snapshots.** VirtualBox can save arbitrary snapshots of the state of the VM. You can go back in time and revert the VM to any such snapshot and start an alternative VM configuration from there, effectively creating a whole snapshot tree.

- **VM groups.** VirtualBox provides a groups feature that enables the user to organize and control VMs collectively, as well as individually.

- **Clean and modular architecture.** VirtualBox has an extremely modular design with well-defined internal programming interfaces and clean separation of client and server code (i.e. code related to VMs and code related to the VMM, respectively).

2.2 VirtualBox architecture

In this section we will present the internal architecture of VirtualBox, giving details about its implementation that is necessary to understand in order to implement our optimizations.

2.2.1 VirtualBox components

When the VirtualBox Graphical User Interface (GUI) is opened and at least a VM is started, three processes are running:

- *VBoxSVC* is the VirtualBox service that always runs in background. This process is started automatically by the first client process [1] and exits a short time after the last client exits. The service is responsible for maintaining the state of all VMs. It is also called *server* process
- The GUI process. It communicates settings and state changes to *VBoxSVC*. It is also called *client* process
- The hypervisor process.

When we launch VirtualBox (e.g. the VirtualBox GUI), the VBoxSVC component starts executing and initializes all the registered VMs. Then, when a VM is launched, before it starts executing, the client asks the server all settings the details, so that it can deliver these information to the starting VMM process. Finally, the hypervisor is able to execute the VM.
In section 3.1.3 we will see these steps in details.

2.2.2 VirtualBox kernel modules

VirtualBox provides different kernel modules that the user should add to the host kernel:

- *vboxdrv*: The only mandatory module. This is needed by the VMM to gain control over the host system. It is used to manage the host/guest world switches and device emulations.
- *vboxnetadp*: "vboxnetadp" stands for "VirtualBox Network Adapter". It is needed to create a host networking interface (called *vboxnet*); that interface (basically a virtual switch), is used to connect VMs to each other (and/or to the host). It is necessary when VirtualBox networking mode is set on **host-only** or **bridged** (these modes will be explained in section 2.2.6).
- *vboxnetflt*: "vboxnetflt" stands for "VirtualBox Network Filter". It is a kernel module that attaches to a real interface on the host and filters and injects packets. As for vboxnetadp, it is only necessary for host-only and bridged modes.

[1] i.e. a frontend. See section 3.1.2 for more informations.

- *vboxpci*: This kernel module provides PCI card passthrough. This is used when the user wants to use a PCI device on the guest, even if the related driver is not available on the host.

2.2.3 Software Virtualization

As stated in section 2.1, VirtualBox fully supports hardware virtualization. However, since this is not a requirement, in the event that hardware virtualization is not present, VirtualBox makes use of a technique defined as *Software virtualization*. In order to understand the software virtualization technique, it is important to understand how CPUs provide a mechanism of protection at microcode level called *Protection rings*.

Privilege rings

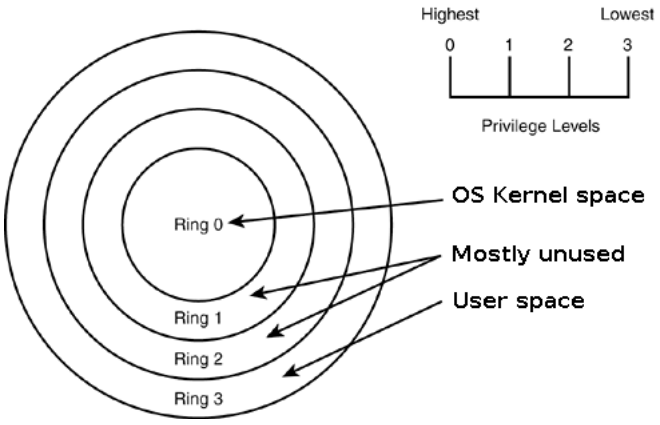

Figure 2.1: Protection rings

As shown in figure 2.1, there are four privilege levels or *rings*, numbered from 0 to 3, with ring 0 (R0) being the most privileged and ring 3 (R3) being the least. The use of ring allows for system software to restrict task from accessing data or executing privileged instructions. In most environments, the OS and some device drivers run in R0 and applications run in ring 3 [9].

Software Virtualization

In addition to the four privilege rings provided by the hardware, we need to differentiate between *host context* and *guest context*.

- In *host context*, everything is as if no VMM was active. This might be the active mode if another application on the host has been scheduled CPU time; in that case, there is a host R3 mode and a host R0 mode.

- In *guest context* a VM is active. So long as the guest code is running in ring 3, this is not much of a problem since a hypervisor can set up the page tables properly and run that code natively on the processor. The problems mostly lie in how to intercept what the guest's kernel does.

When starting a VM, VirtualBox sets up the host system through its R0 support kernel driver so that it can run most of the guest code natively, and it inserts itself at the ”bottom” of the picture. It can then assume control when needed, e.g. if a privileged instruction is executed, the guest *traps*; VirtualBox may then handle this and either route a request to a virtual device or possibly delegate handling such things to the guest or host OS. In guest context, VirtualBox can therefore be in one of three states:

- Guest R3 code is run unmodified, at full speed, as much as possible. The number of faults will generally be low. This is also referred to as *raw mode*, as the guest R3 code runs unmodified.

- For guest code in R0, VirtualBox employs a trick: it actually reconfigures the guest so that its R0 code is run in **ring-1** (R1) instead. As a result, when guest R0 code (actually running in R1) such as a guest device driver attempts to write to an I/O register or execute a privileged instruction, the VirtualBox hypervisor in ”real” R0 can take over.

- The VMM can be active. Every time a fault occurs, VirtualBox looks at the offending instruction and can relegate it to a virtual device, the host OS, the guest OS, or run it in the **recompiler**.

In particular, the recompiler is used when guest code disables interrupts and VirtualBox cannot figure out when they will be switched back on. The recompiler is based on the dynamic translation technique (section 1.1.2).

Unfortunately this only works to a degree. Among others, the following situations require special handling:

1. Running R0 code in R1 causes a lot of additional instruction faults, as R1 is not allowed to execute any privileged instructions (of which guest's R0 contains plenty). With each of these faults, the VMM must step in and emulate the code to achieve the desired behavior. While this works, emulating thousands of these faults is very expensive and severely hurts the performance of the virtualized guest.

2. There are certain flaws in the implementation of R1 in the x86 architecture that were never fixed. Certain instructions that *should* trap in R1 don't. If the guest is allowed to execute these, it will see the true state of the CPU, not the virtualized state.

3. A hypervisor typically needs to reserve some portion of the guest's address space for its own use. This is not entirely transparent to the guest OS and may cause clashes.

4. The SYSENTER instruction (used for system calls) executed by an application running in a guest OS always transitions to R0. But that is where the VMM runs, not the guest OS. In this case, the hypervisor must trap and emulate the instruction even when it is not desirable.

5. The CPU segment registers contain a "hidden" descriptor cache which is not software-accessible. The hypervisor cannot read, save or restore this state, but guest OS may use it.

6. Some resources must (and can) be trapped by the hypervisor, but the access is so frequent that this creates a significant performance overhead.

To fix these performance and security issues, VirtualBox contains a *Code Scanning and Analysis Manager* (CSAM), which disassembles guest code, and the *Patch Manager* (PATM), which can replace it at runtime.

Before executing R0 code, CSAM scans it recursively to discover problematic instructions. PATM the performs *in-situ* patching, i.e. it replaces the instruction with a jump to hypervisor memory where an integrated code generator has places a more suitable implementation. In reality, this is a very complex task as there are lots of odd situations to be discovered and handled correctly.

In addition, every time a fault occurs, VirtualBox analyzes the offending code to determine if it is possible to patch it in order to prevent it from causing more faults in the future. This approach works well in practice and dramatically improves software virtualization performance.

2.2.4 Hardware virtualization

As stated in section 1.1.3, with Intel VT-x there are two distinct modes of CPU operation: root mode and non-root mode.

- In root mode, the CPU operates much like older generations of processors without VT-x support. There are four privilege rings, and the same instruction set is supported, with the addition of several virtualization specific instructions. Root mode is what a host operating system without virtualization uses, and it is also used by a hypervisor when virtualization is active.
- In non-root mode, CPU operation is significantly different. There are still four privilege rings and the same instruction set, but a new structure called *Virtual Machine Control Structure* (VMCS) now controls the CPU operation and determines how certain instructions behave. Non-root mode is where guest systems run.

The VMCS includes a guest and host state area which is saved/restored when switching between the two modes (*VM entry* and *VM exit*). Most importantly, the VMCS controls which guest operations will cause VM exits. Thanks to the

VMCS, a hypervisor can allow a guest to write certain bits in shadowed control registers,but not others. This enables efficient virtualization in cases where guests can be allowed to write control bits without disrupting the hypervisor, while preventing them from altering control bits over which the VMM needs to retain full control. The VMCS also provides control over interrupt delivery and exceptions.

Whenever an instruction or event causes a VM exit, the VMCS contains information about the exit reason. Thus the hypervisor can efficiently handle the condition without needing advanced techniques such as CSAM and PATM described above.

VT-x inherently avoids several of the problems which software virtualization faces. The guest has its own completely separate address space not shared with the hypervisor, which eliminates potential clashes. Additionally, guest OS kernel code runs at privilege R0 in non-root mode, obviating the problems by running R0 code at less privileged levels. Naturally, even at R0 in non-root mode, any I/O access by guest code still causes a VM exit, allowing for device emulation.

We restrict our work to the Hardware Virtualization solutions, because this is the one that maximizes overall performance.

2.2.5 Emulation Threads

When a VM is started, a user-defined number of SMP processors is assigned to the VM itself. Each CPU is emulated through a so called *Emulation Thread* (EMT), so we have one EMT per SMP processor. An EMT is responsible of executing guest code, emulating devices and handle the transition between host world and guest world.
When using hardware virtualization (as we stated before, this is our case), an EMT continuously switches between root mode and VM mode (see section 1.1.3).

Let us assume that the EMT is running guest code, e.g. an application. At

some point, the application tries to execute an I/O operation, causing a SYSENTER instruction in the guest. Therefore, the CPU executing the EMT, switches from R3 to R0 (still in VM mode), in order to run the guest kernel code. At this point, the ”true” I/O operation (such as a write operation in a register) produces a VM exit, so the CPU switches from VM mode to root mode. On a VM exit the EMT stops executing guest code and start executing VirtualBox code (in kernelspace), in order to handle the event that caused the VM exit itself. Handling a VM exit *may* cause the EMT to execute userspace code in order to emulate a device, switching back to kernelspace after the device emulation (more on this in section 2.4). After the event has been handled, the EMT executes a VM entry (i.e. CPU switches back to VM mode) and continues to run guest code from the point where it was interrupted.

Figure 2.2 shows the situation described above.

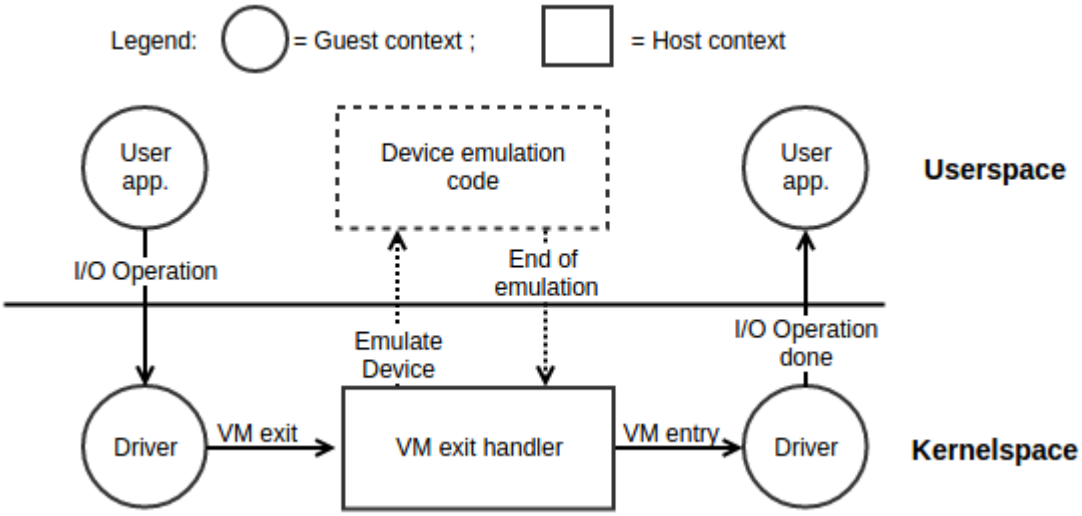

Figure 2.2: Example of EMT execution flow

2.2.6 Networking modes

For a VM it is fundamental to communicate to the outside world using the networking infrastructure, otherwise the VM becomes useless.
Nevertheless, a VM is ”just” a software entity, so it is not connected to any *real*

network. Therefore the VMM must provide some kind of virtualized network infrastructure, so that guest OS thinks its **virtual** network device is connected to a **physical** network and can then exchange packets with the outside.

VirtualBox provides several of these network infrastructures called *Networking modes*; in that way a user can choose the most suitable way to connect her VM. Among the others, the main network modes provided by VirtualBox are the following:

- **Not attached** In this mode, VirtualBox reports to the guest that a network card is present, but there is no connection, as if no Ethernet cable was plugged into the card.

- **Network Address Translation (NAT)** A VM with NAT enabled acts much like a real computer that connects to the Internet through a router. The ”router”, in this case, is the VirtualBox networking engine, which maps traffic from and to the VM transparently. In VirtualBox this router is placed between each VM and the host.
  The VM receives its network address and configuration on the private network from a DHCP server integrated into VirtualBox. The IP address thus assigned to the VM is usually on a completely different network than the host.

- **Bridged networking** With bridged networking, VirtualBox uses a device driver on the host system (*vboxnetflt* kernel module) that filters data from the physical network adapter. That is why it is called ”network filter” driver. This allows VirtualBox to intercept data from the physical network and inject data into it, effectively creating a new network interface in software. When a guest is using such a new software interface, it looks to the host system as though the guest were physically connected to the interface using a network cable: the host can send data to the guest through that interface and receive data from it. This means that the user can set up routing or bridging between the guest and the rest of the network.

- **Internal networking** It is similar to bridged networking in that the VM can

directly communicate with the outside world. However, the ”outside world” is limited to other VMs on the same host which connect to the same internal network, which is identified simply by its name.
Even though technically, everything that can be done using internal networking can also be done using bridged networking, there are security advantages with internal networking. In bridged mode, all traffic goes through a physical interface of the host system. It is therefore possible to attach a packet sniffer to the host interface and log all traffic that goes over it. If, for any reason, the user prefers two or more VMs on the same machine to communicate privately, hiding their data from both the host system and the user, bridged networking therefore is not an option.

- **Host-only networking** Host-only networking can be thought as a hybrid between the bridged and internal networking modes: as with bridged networking, the virtual machines can talk to each other and the host as if they were connected through a physical Ethernet switch. Similarly, as with internal networking however, a physical networking interface need not to be present, and the virtual machines cannot talk to the world outside the host since they are not connected to a physical networking interface.
  Instead, when host-only networking is used, VirtualBox creates a new software interface on the host (using *vboxnetadp* kernel module) which then appears next to the existing network interfaces. In other words, whereas with bridged networking an existing physical interface is used to attach virtual machines to, with host-only networking a new ”loopback” interface is created on the host. And whereas internal networking the traffic between the VMs cannot be seen, the traffic on the ”loopback” interface on the host can be intercepted.

NAT mode is not interesting with respect to our goals, since it is only intended to be a way the VM can easily access the Internet, and it is not intended to be an efficient networking mode. Similarly, we will not consider bridged networking mode, because optimizing the performance of a real network adapter is not the aim of this work. Instead, we will consider the host-only mode, since our goal is to optimize the communication performances between two VMs on the same

host, or between a VM and the host (so also internal networking is not interesting for us), using the netmap framework.

2.2.7 Network port and connector

In order to implement a specific networking architecture, VirtualBox implementation includes an interface between the code that emulates the network adapter, and the code that provides access to the chosen networking mode. The reason is that the two subsystems are completely independent, and a user can easily combine every virtual network adapter with every networking mode.
VirtualBox defines the network device emulation *network port* and the networking mode *network connector*.

Ports and connectors are two interfaces that communicate via a callback mechanism. In any way a code implements a port interface, it must have a reference (i.e. a pointer) to a connector, and vice versa.

Ports The methods [2] exported by ports (which are exposed to connectors) are the following:

- `pfnWaitReceiveAvail` Waits until there is space for receiving data. It also takes the number of milliseconds to wait (timeout) as argument. If timeout parameter is 0, then it returns immediately. The return value specifies if there is space available to receive data, if timeout expired or if an error occurred.

- `pfnReceive` When the connector receives data from the network, it calls this function on the port, so the latter can push data to the guest. This function takes the pointer to available data and the number of bytes in the buffer as arguments.

[2] The ”pfn” prefix stands for ”pointer to function”. It is imposed by VirtualBox coding guidelines.

- `pfnReceiveGso` The same as `pfnReceive`, but it has and additional argument regarding the *segmentation offloading* context [3].

- `pfnXmitPending` This function is used to notify the port that can transmit pending packets (if any).

Connectors The methods exported by connectors (which are exposed to ports) are the following:

- `pfnBeginXmit` It is used by port to get a lock on the connector (only one port instance can transmit at a time).

- `pfnAllocBuf` This asks the connector to provide the buffer that the port will fill with data. The size of that buffer is specified as an argument.

- `pfnFreeBuf` Frees an unused buffer that has been requested by the port.

- `pfnSendBuf` Sends data to the network. After the port filled the buffer (allocated by the connector) with data, it calls this function passing the filled buffer to the connector.

Moreover, both interfaces export two other (mandatory) methods: `pfnConstruct` (constructor) and `pfnDestruct` (destructor). The constructor is called when the VMM instantiates a port/connector, the destructor is called during the shutdown process.

When a port wants to send a frame to the network, it invokes the `pfnBeginXmit` function provided by the connector in order to gain lock access on the connector itself; then it calls `pfnAllocBuf` function to get a buffer where the port can store the frame and, finally, it invokes `pfnSendBuf` so that the connector can push the frame (passed as argument) to the network. On the other direction, when the connector gets a frame from the network, it invokes the `pfnWaitReceiveAvail` to check whether the port is able to receive data. If this is the case, it immediately calls `pfnReceive` so that the port can push the received frame (passed as

[3] We are not going into details, since in our work we did not implement segmentation offloading.

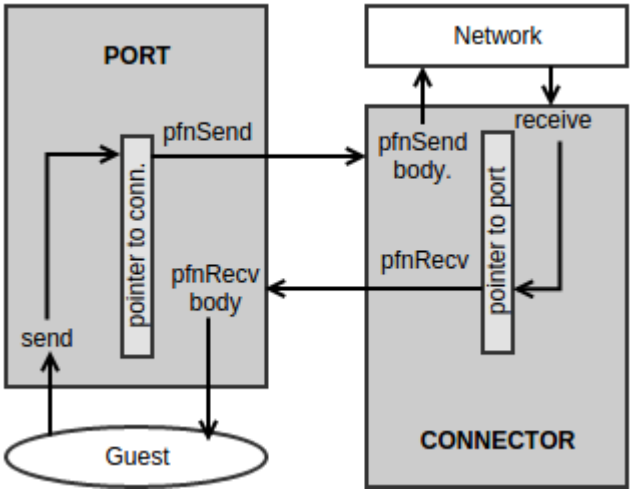

Figure 2.3: Example of EMT execution flow

argument) to the guest.
A simplified version of the above interaction is depicted in figure 2.3, where only the *send* and *receive* functions are shown.

2.3 The e1000 network adapter

In this section we will outline of the functioning of an e1000 device. We will describe only those aspects that are relevant to our goals, in particular we are interested to the NIC *datapath*. The complete specifications can be found at [10]

The datapath refers to the software interface that the OS uses in order to transmit and receive Ethernet frames. It involves just some registers and DMA-mapped memory.
When the device driver wants to send an Ethernet frame through the adapter, it has to tell the adapter where the frame is stored in physical memory and how long it is. Once the device is aware of where the frame is stored, it can directly access the physical memory and send the frame on the wire. Similarly, when a frame arrives from the wire, the adapter has to store it in the physical memory. For this reason, it must know in advance where to store the frames, so the device driver must tell the adapter where it can store arrived frames. If it is not the case, the

adapter will drop incoming frames.

As we can see, in order to achieve this information exchange, there must be a well-defined interface between the device driver and the adapter. This interface is known as a *ring*. A ring is a circular **array** (i.e. a contiguous zone in memory) of *descriptors* that are used to exchange those information. A network device has at least two rings: one for transmission (*TX ring*) and the other for reception (*RX ring*). A network adapter can have multiple TX/RX rings, possibly with different priorities and/or policies, so that it permits traffic engineering.
However, VirtualBox implementation of e1000 device, offers only one ring per direction. The number of descriptors per ring (i.e. the length of the array), can be chosen by the device driver. In e1000 this number must be a power of 2 and less than or equal to 4096.

2.3.1 TX ring

The TX ring is an array containing *N* TX descriptors. Each descriptor is 16 bytes long and contains the physical address of the associated buffer, its length (i.e. the length of the stored frame) and some status flags. Among the others, it contains the *Descriptor Done* flag (DD) and the *End of OPeration* flag (EOP). The DD flag is set by the adapter to tell the device driver that the TX descriptor has been successfully processed. The EOP flag is used by the device driver to tell the adapter whether that TX descriptor contains a complete packet or only a part of it (e.g. because it does not fit in the buffer); so if a packet is spread among *N* TX descriptors, the first *N-1* descriptors will have the EOP flag set to 0, and the last one will have the flag set to 1.

Since the ring is stored in physical memory, the adapter must know its physical address. This information is stored in two registers: TDBAL (*Transmit Descriptor Base Address Low*) and TDBAH (*Transmit Descriptor Base Address High*). These are two 32-bit registers that, concatenated, form a 64-bit string of bits which is the physical base address of the ring.
Since it is a producer/consumer system, a synchronization mechanism is required

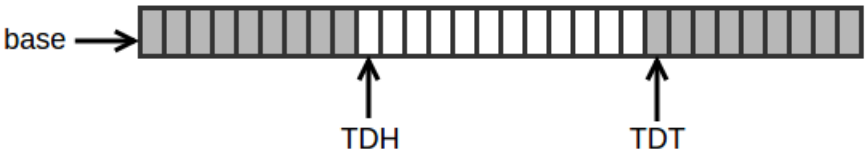

Figure 2.4: TX ring with its registers. Free descriptors are the grey ones, while the pending descriptors are the white ones. *base* is the concatenation of TDBAH and TDBAL registers.

between the device driver and the adapter. This is achieved using two *index registers*: the TDT register (*Transmit Descriptor Tail*) and the TDH register (*Transmit Descriptor Head*). The value of these registers represent an *array index* with respect to the TX ring.

At the beginning, TDT and TDH are initialized to their initial value (0) by the device driver. When the driver wants to send a new frame, it writes the physical address and the length of the frame in the descriptor pointed by TDT register, then it increments the TDT register itself (modulo number-of-descriptors).
When the adapter recognizes that TDH and TDT are different, it understands that there are new frames to be sent on the wire, so it start processing the descriptors starting from the one pointed by TDH.
For each new descriptor to process, the device:

1. Sends the new frame on the wire.
2. Writes the TX descriptor back in order to set the DD flag to 1.
3. Increments the TDH register circularly.

So the adapter can access the next descriptor to be processed with this formula:

$$Index = base + (TDH \times 16)$$

Where *base* is the concatenation of TDBAH and TDBAL registers.

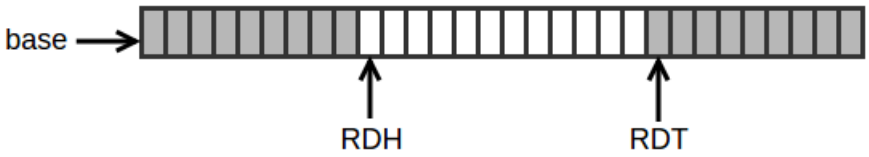

Figure 2.5: RX ring with its registers. Filled descriptors are the grey ones, while the available descriptors are the white ones. *base* is the concatenation of RDBAH and RDBAL registers.

The **adapter** stop condition is $TDH == TDT$, i.e. there are no more descriptors to process (TX ring empty).
In order to prevent the index registers to wrap around, the device driver must never use the last free descriptor. So when the TDT is such that incrementing it circularly would cause $TDH == TDT$, the driver must stop transmitting. This is the *TX ring full* condition.
Figure 2.4 shows the TX ring with its registers.

2.3.2 RX ring

The RX ring is an array of *N* RX descriptors. Each RX descriptor is 16 bytes long and contains the physical address of the associated buffer, its length and some status flags. Among the others, it contains the DD flag and the EOP flag (just like a TX descriptor). The DD flag is used by the adapter to notify the device driver that a the RX descriptor contains new data to be processed. The EOP flag is used by the adapter to tell the device driver whether that RX descriptor contains a complete packet or only a part of it. So if a packet is spread over *N* RX descriptors, the first *N-1* will descriptors will have the EOP flag set to 0, while the last one will have the flag set to 1.

Just as for TX ring, the adapter must know the physical address of the RX ring. This address is stored in two registers: RDBAL (*Receive Descriptor Base Address Low*) and RDBAH (*Receive Descriptor Base Address High*). These are two 32-bit registers that, concatenated, form a 64-bit string of bits which is the physical base address of the RX ring.

Here also, the synchronization between the adapter and the device driver is implemented through two index registers: RDT register (*Receive Descriptor Tail*) and RDH (*Receive Descriptor Head*).

At the beginning, the driver initializes RDH and RDT to their initial values (0). The adapter still does not know the physical address of any buffer where it can store frames, so it cannot store incoming frames. Therefore, the driver writes the physical address of a memory buffer into the RX descriptor pointed by RDT, and increment RDT circularly[4]. Now the device know that a new memory buffer is available, so it use it to store incoming frames. When $RDH \neq RDT$, the adapter knows that there are buffers available, so it can accept incoming frames.
When a new frame is arrived from the wire, the adapter:

1. Fetches the RX descriptor pointed by the RDH register.
2. Copies the frame to the buffer pointed by the fetched RX descriptor.
3. Writes back the descriptor in order to set the length of the received frame and the DD bit (*writeback*).
4. Increments RDH register circularly.
5. *May* send an interrupt in order to tell the driver that a new frame is available to be delivered to the network stack[5].

So the adapter can access the next available RX descriptor with this formula:

$$Index = base + (RDH \times 16)$$

where *base* is the concatenation of RDBAH and RDBAL registers.

The **adapter** stop condition is $RDH == RDT$, i.e. there is no available free descriptor where a frame can be stored.
As for the TX ring, in order to prevent the array indexes to wrap around, the

[4]It is useless to write the length field in the descriptor, since it will be modified by the adapter on reception.

[5]See section 2.4.2.

driver should never increment RDT if the increment would cause $RDT == RDH$. This is the full RX ring condition.
Figure 2.5 shows the RX ring with its registers.

2.3.3 e1000 interrupts

The e1000 network adapter can generate interrupts for several reasons, but we are only interested in two of them:

- *TX interrupts*: these interrupts are raised when the transmission of one or more frames completed. Each TX descriptor has a bit flag *Report Status* (RS), that, if set, tells the adapter to raise an interrupt as soon as the associated frame is transmitted. Anyway, an interrupt is always sent when the adapter reaches the stop condition ($TDT == TDH$).
  The interrupt handler frees the descriptors that have been processed (DD flag set), and mark them as free (unset the DD flag).

- *RX interrupts*: these are raised whenever the adapter stores a new incoming frame in physical memory; in that way the device driver knows that a new frame has been received and it can be pushed to the kernel network stack.

When we are dealing with high packet rates, e.g. 1 Mpps, interrupt rate becomes a critical issue. In fact, interrupt routines have a fixed cost that must be paid before doing useful work, such as push the received frame to the network stack and let the receiver application to process it.
At this rate, if each received packet raised a RX interrupt, the we would handle up to 1 million of interrupts per second, which would stall the machine. In that case, the CPU would spend almost all of its time in handling interrupts, and the receiver application could not do any useful work. This problem is known as the *livelock* problem.

This problem can be solved if the device ”skips” some RX interrupts, raising an interrupt every batch of received frames, e.g. 100 frames per batch, and not every single frame. In that way, the interrupt rate is 100 lower and the interrupt overhead is amortized over 100 frames. Anyway, the device must guarantee that

a RX interrupt is raised after a period of time, even if the 100-frames batch is not completed, because the device cannot know when the next frame will arrive.
These mechanisms are known as *interrupt mitigation*.
The e1000 network adapter implements two interrupt mitigation mechanism, but since the older one is strongly discouraged by the Intel manual, we will consider only the most recent one.

The e1000 network adapter has a register, called *Interrupt Throttling Register* (ITR), which controls the interrupt mitigation mechanism. If the driver sets this register to a value of δ, the hardware ensures that δ is the minimum inter-interrupt interval, regardless of the interrupt type. In other words, whenever an event that requires an interrupt occurs, such as TX completion or RX completion, the device raises an interrupt as soon as possible, while meeting the ITR inter-interrupt delay constant.

2.4 VirtualBox e1000 emulation

The e1000 port is implemented in VirtualBox through three source files [6]: *DevE1000.cpp*, which is the one we are interested in, *DevE1000Phy.cpp* and *DevEEPROM.cpp*. The first one contains all the "emulation logic" part, while the others implement only the internal physical emulation and the internal EEPROM respectively.
The code is an implementation of the interfaces provided by VirtualBox. As mentioned in section 2.2.7, the first and last called method are the constructor (`pfnConstruct`) and the destructor (`pfnDestruct`) respectively. The main purpose of these functions is to register/deregister the new PCI Ethernet device with the rest of the emulator. In this way, it is possible to have multiple instances of the e1000 network device when launching VirtualBox.
Furthermore, the code contains (in the first part of the file), a set of *options* , implemented as `define` statements, that can be enabled/disabled before compiling (e.g. enable usage of caches, ITR register, and so on).

[6] These source files are located in src/VBox/Devices/Network/ in the VirtualBox project root directory.

When registering a new PCI device, it is necessary to describe the I/O or MMIO regions that the device implements the device: this is done registering callback functions to to those regions, one for *in* operations, on for *out* operations. The e1000 emulation code registers a MMIO region and an I/O region, but the latter is not used. The MMIO region maps all the register the e1000 device implements.
A statically defined mapping table is used to associate a couple of functions to each register, one for *IN* operation, one for *OUT* operation. In that way, one may associate a different read callback and a different write callback for each e1000 register. Is also possible to have the same callback function for multiple registers, or have no callbacks for some of them.
In short, the emulation of a device is achieved with pre-registered callbacks.

Now we will see in details how the register callbacks are invoked.
When an EMT is executing guest code, e.g. the e1000 device driver, it may try to access a MMIO location corresponding to an e1000 register. The accessing instruction causes a VM exit, so the EMT switches from the guest context to the host context. At this point, the VirtualBox driver analyzes the VM exit reason and understand that the VM exit was caused by an MMIO access. In our case, the callback registered with the e1000 MMIO is invoked. This callback uses the address to get the index of the accessed register and calls the read (write) handler specific for that register.
After the callback returns, a VM entry is executed and the EMT resumes executing guest code.

Event queues

A register callback is invoked by the EMT (section 2.2.5) while it is handling the VM exit event. So we are in R0 context. That said, writing a register may cause some side effects, therefore these side effects should be emulated. However, since it may take an amount of time that is, possibly, much longer than a simple register update, and also because it is unnecessary to execute that code in kernelspace, it

is reasonable to execute the ”side-effect” code in userspace.
In order to achieve this, VirtualBox provides the so called *Event queues*. As the name suggests, these are queues in which an *event* can be posted. Each queue has a callback function that is associated to the queue itself.
Every time the EMT finishes handling a VM exit it checks all the event queues before executing a VM entry. If an event is found, the EMT removes that event from the queue and switches from R0 context to R3 context, so that it can start executing the callback associated to that queue.
When the callback returns, the EMT switches back to the kernelspace and continues scanning all event queues. Finally, when there are no more pending events, it executes a VM entry and resumes executing guest code.

2.4.1 TX emulation

The TX execution path is performed by the EMT thread. As described in section 2.3, when the device driver wants to notify the hardware that a new TX frame is ready to be processed, it writes to the TDT register. This causes a VM exit to occur, so the EMT passes from guest context to host context.
Writing to the TDT register causes the adapter to transmit one or more frames, so the transmission handling must be done in userspace. So the write callback for the TDT register, aside from updating the register value, posts an event to an event queue (*TxQueue*); in this way the EMT, before switching back to the guest world, executes the transmission of frames in userspace.
The callback associated with the event queue simply calls the `e1kXmitPending` function. The way this function is implemented is regulated from the `E1K_WITH_TXD_CACHE` define.

When TX descriptors cache is disabled, the `e1kXmitPending` function:

1. *Tries to acquire the lock* It calls the `pfnBeginXmit` function provided by the connector, where it **tries** to acquire the lock on the connector itself. This is a so called *trylock* function, because if the lock is busy, the thread does not block waiting for the lock to be acquired. Instead, the function returns a *busy* value. This is necessary because the thread trying to get the

lock is the EMT, so it **must never** block, since it is in charge of emulating all the system. If the trylock succeeds, then go to the next step, otherwise the function returns without doing any transmission.

2. *Loads the current descriptor* If there are available descriptors ($TDT \neq TDH$), then it reads the TX descriptor pointed by TDH register from the guest physical memory.

3. *Allocates buffer* The port asks the *connector* to allocate the buffer where it can store the frame, which is pointed by the loaded TX descriptor. The information about the length of the frame is contained in the TX descriptor itself.

4. *Writes back the descriptor* It writes the descriptor back into the guest physical memory, setting the DD bit to 1.

5. *Copies the frame* It reads the physical address pointed by the current descriptor (where the frame is stored) and copies it into the buffer provided by the connector.

6. *Possibly Transmits frame* If the packet is complete, i.e. EOP flag of TX descriptor is set to 1, then it calls the `pfnSendBuf` provided by the connector, passing the previously allocated buffer as an argument.

7. *Updates register* It circularly increments the TDH register.

8. *Possibly raises an interrupt* If at least one complete packet has been sent, it *may* raise an interrupt to notify the guest that one or more frames have been sent (see section 2.3.3). If there are other descriptors available ($TDT \neq TDH$), then go back to step 2.

9. *Releases the lock* After all operations, it releases the lock acquired at step 1 calling the `pfnEndXmit` provided by the connector.

We can see that for each available descriptor, the EMT must read the guest physical memory in order to get the current TX descriptor. This introduces an high overhead, particularly when the number of available descriptors is much

higher than 1.

The reading overhead can be amortized using a local TX descriptor cache. This can be done enabling the related option (`define E1K_WITH_TXD_CACHE`).
When the cache is enabled, the `e1kXmitPending` function behaves much like the same as before, except for the step 2. Previously, we had one TX descriptor per read. Now, instead, the EMT **tries** to load all the available TX descriptors in one single physical read (two reads in case the tail wrapped around the end of the TX descriptor ring). However, it may happen that only a fraction of the available descriptors is loaded, e.g. the cache is almost full. In that case, after the partial loading, the fetched descriptors are processed (like the previous case), the frames are sent and the cache is flushed, so that other available descriptors can be loaded. In that way, the cost of the readings is amortized over the number of fetched descriptors.

2.4.2 RX emulation

This time the RX execution path is not performed by the EMT, but it is performed by another thread. The reason is that the EMT is in charge of the entire emulation process, so it must never do blocking operations, so it cannot wait for incoming frames from the network.
The thread in charge of receive frames is dependent on the used connector: in fact the frames flow from the network to the connector, and the latter sends them to the port (see section 2.2.7, figure 2.3).
From now on, we will refer to this thread as *recv thread*.

When one or more frames from the network, the recv thread stores them in a buffer (that is dependent from the connector implementation). Then it first calls the port callback `pfnWaitReceiveAvail` to check if the port is able to receive at least one frame. If that callback returns successfully, the thread calls the `pfnReceive` callback, passing the buffer with the available data.

The e1000 port implements the `pfnReceive` method with

`e1kR3NetworkDown_Receive` function, which takes the buffer and the size of the available data in the buffer as arguments.
As in the TX case, the behavior (i.e. the implementation) is controlled by the `E1K_WITH_RXD_CACHE` define.

When RX descriptor cache is disabled, this function:

1. *Filters the packet* It determines if the packet is to be delivered to the upper layer. The decision is based on:

   - Length: if the packet length is greater than the maximum supported size (16384 bytes) or if long [7] packet reception is disabled [8], then drop the packet.
   - VLAN tag: if the filter does not find a match for the VLAN tag, then drop the packet.
   - Exact Unicast/Multicast: if the packet destination address exactly matches the address (either unicast or multicast address), then deliver the packet, otherwise drop it.
   - Promiscuous Unicast/Multicast: if the adapter is set in promiscuous mode [9], then deliver the packet, otherwise drop it.
   - Broadcast: if the packet destination address is broadcast, deliver it.

2. *Pads the packet* If the received packet is too short (less than 60 bytes), the packet is padded with zeroes.

3. *Loads the RX descriptor* If there is at least one available RX descriptor ($RDT \neq RDH$), then it loads the RX descriptor pointed by RDH register from the guest physical memory.

4. *Fills the buffer* After loading the RX descriptor, it writes the (possibly padded) packet into the buffer pointed by the loaded RX descriptor. If the

[7] Packet greater than 1522 bytes.
[8] For further information, see [10].
[9] The adapter accepts packets even if they are not addressed to it

packet does not fit the buffer, the EOP flag of the RX descriptor is set to 0, otherwise the flag is set to 1.

5. *Writes back the RX descriptor* After the RX descriptor has been filled with data, it sets the DD bit to 1 and writes back the RX descriptor in memory.

6. *Updates register* It circularly increments the RDH register.

7. *Possibly raises an interrupt* If at least one complete packet is successfully delivered, it *may* raise an interrupt to notify the guest that one or more frames have been received (see section 2.3.3). If there is still data to be processed (e.g. a packet is not completely stored), go back to step 3.

When the reception is completed (i.e. the `pfnReceive` function returns), the recv thread goes back to sleep waiting for new incoming packets.

Just as in the TX case, we can see that for each available descriptor, the recv thread must read the guest physical memory to get the current RX descriptor. Therefore we have an high overhead, in particular when we are dealing with large incoming packets.

This overhead can be amortized using a local RX descriptor cache, that can be enabled through the related option (`define E1K_WITH_RXD_CACHE`). When the cache is enabled, the `e1kR3NetworkDown_Receive` function behaves much like before, except for what concerns the loading of RX descriptors. In the previous case, we had one single RX descriptor per read. Now the recv thread, instead of loading the descriptor in memory, looks at the local cache: if the cache is empty, it *prefetches* a number of RX descriptors, which is the minimum between the cache size and the number of available RX descriptors in memory, with a single memory read (two reads in case the tail wrapped around the end of the RX ring). If the cache is not empty, it uses the first available RX descriptor in cache.
After the RX descriptor is processed and written back, the related position in cache is cleaned.

It is important to point out that the physical **guest** memory is different from the physical **host** memory. Therefore, all the accesses to the guest physical memory (such as TX/RX descriptor loads) require an *address translation*, that introduces additional overhead. The way it can be obviated will be described in section 4.3.

Chapter 3

Implementation of netmap support

In chapter 2 we described the VirtualBox architecture, its implementation and the interfaces it provides. In this chapter we will implement an extension to VirtualBox that provides fast packet I/O. This is achieved by interfacing VirtualBox with the netmap framework [1].

As described in section 2.2.7, VirtualBox includes two interfaces in order to implement the networking infrastructure: *ports* and *connectors*. Our goal is to create a new connector that implements the interfaces a VirtualBox port to a VALE switch [3] provided by the netmap framework.
Our netmap connector will be implemented through one single source file: *DrvNetmap.cpp*[1].

3.1 Integration with VirtualBox

Before going into details of our implementation, we must first ”integrate” our work in VirtualBox system. It basically involves the VirtualBox build process, the user interface, so that a user can choose netmap as connector for his system, and the VMs configuration mechanism, in order to set/save/load the new configuration parameters needed by netmap.

[1]This source file will be located in src/VBox/Devices/Network/ in the VirtualBox project root directory. The *Drv* prefix in the filename is due to the VirtualBox coding guidelines.

3.1.1 Building VirtualBox

The VirtualBox build process is performed through two steps: the configuration and the compilation.

Configuration

Before starting the compilation process, we must run the script *configure*[2], in order to check if the system meets the requirements, e.g. the presence of needed libraries, and to let the user to specify some custom options (e.g. the `--disable-docs` option prevents the compilation of documentation files). Therefore, we added the "`--with-netmap=`*dir*" option to that script, so that the user can specify the absolute path [3] of netmap libraries in the system.
The effect of the new option is to set an environment variable that the Makefile will check, so that it knows whether to include or not our *DrvNetmap.cpp* source file in the compilation process, and the netmap headers in the list of directories to be searched for header files[4].

Compilation

As mentioned before, the Makefile will check a netmap-related environment variable that can be enabled through the configuration process. VirtualBox has one *virtual* Makefile, that is actually splitted in many files (more than two hundreds). Those files are organized in a hierarchical manner: there is one Makefile for each level of the source tree. Therefore, the more we go deeper in the source tree, the more specific the present Makefile is for that subtree.
Since our new file will be placed in the network devices directory, we will modify the devices-related Makefile, located in *src/VBox/Devices/* directory.

[2] Located in the root directory of VirtualBox project.

[3] It is requested the path to the subdirectory *sys* of netmap source tree, e.g. `--with-netmap=/path/to/netmap/sys/`.

[4] `-I` option of GNU C compiler.

3.1.2 The VirtualBox User Interface

As stated in section 2.1, VirtualBox offers multiple frontends as user interfaces. We need to extend those interfaces so that a user can choose netmap as connector. We focused on the *VirtualBox* frontend (the default one), and *VBoxManage* frontend, since *VBoxSDL* only launches a VM, but the related configuration is done through VBoxManage.

VirtualBox frontend

This is the default frontend. It offers a user-friendly Graphical User Interface (GUI). The related source files are located in *src/VBox/Frontends/VirtualBox/src*. Since we are interested in the networking part, we will focus on it.
As shown in figure 3.1, VirtualBox frontend allows to specify network settings for

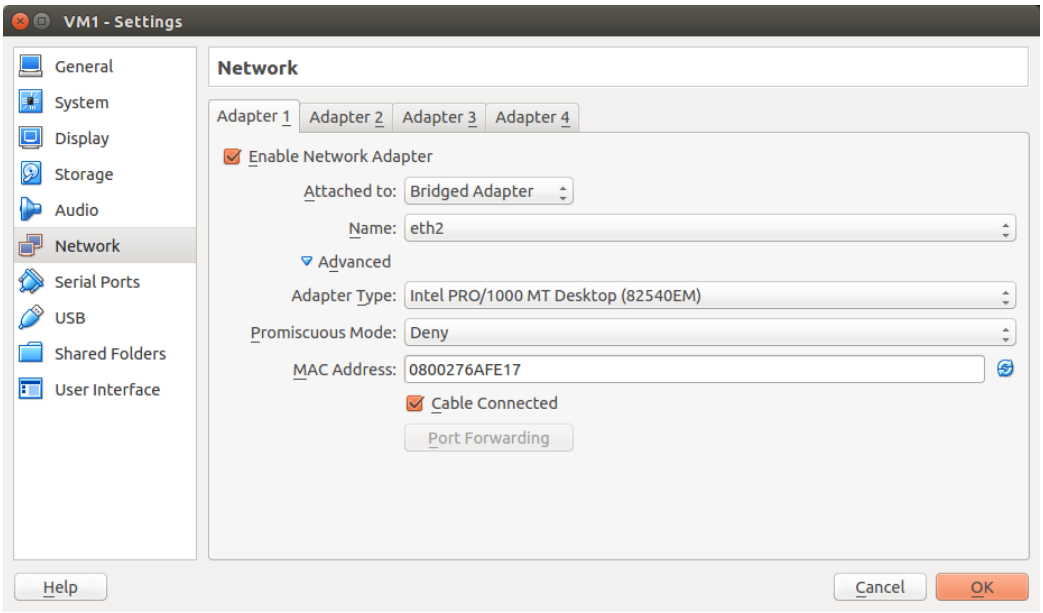

Figure 3.1: Sample of VirtualBox network settings tab. In this case the adapter is an e1000 device, that is bridged to the host interface *eth2*.

each VM independently and, for each VM, to specify up to 4 different network adapters. For each adapter, the following options are available:

- *Enable flag* If checked, the network adapter is enabled in the Guest.

- *Attached to* This drop down menu specifies the connector the adapter will be attached to. See section 2.2.6.
- *Name* It is used in some networking modes. E.g. the bridged mode needs to know which physical interface will be bridged with the emulated interface.
- *Adapter Type* Here we can select the preferred emulated adapter (such as an e1000 device).
- *Promiscuous mode* It is used to set/unset the promiscuous mode on the adapter.
- *MAC Address* The MAC address of the emulated interface.
- *Cable Connected* If enabled, the adapter will be (virtually) connected to the connector.
- *Port Forwarding* Only in NAT mode.

Our goal is to extend this interface so that a user can choose netmap as preferred connector. What we have to do is to include netmap in the "*Attached to*" drop down menu, then add an additional text box where some netmap parameters can be specified by the user.
To achieve this, we modified some source files that define this frontend. They are located in *src/VBox/Frontends/VirtualBox/src/settings/machine*. In that way, we added netmap as an item of the drop down menu and we used the *name* field as a textbox to specify to which netmap port the adapter will be attached to (e.g. a vale switch).

Moreover, we modified the *UIMachineSettingsNetwork.ui* file; this file, which is used by Qt designer tool, describes all the GUI elements in XML (e.g. textboxes, flags, etc). Therefore, we added in the GUI a textbox where the user can specify some netmap parameters, such as the number of netmap rings and the related number of netmap slots [5].
Figure 3.2 shows the result of our modifications.

[5] For further information, refer to [1].

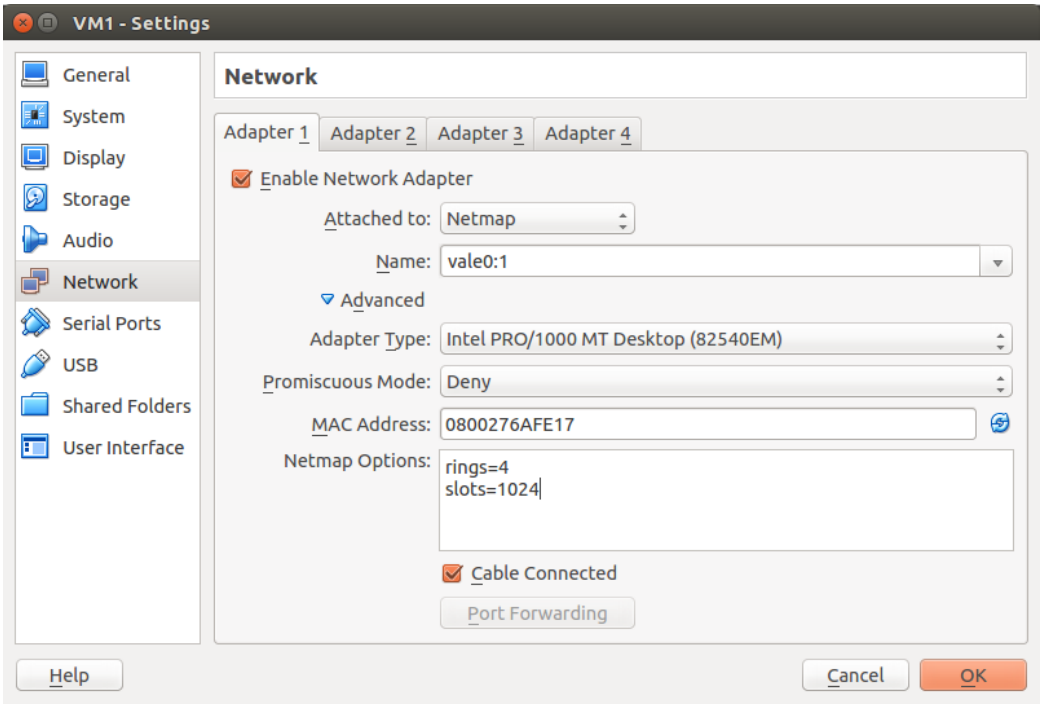

Figure 3.2: The extended VirtualBox network settings interface. It uses the port 1 of the vale0 netmap virtual switch (vale0:1) and specifies 4 netmap rings per direction, having 1024 slots each.

VBoxManage frontend

VBoxManage is a Command Line Interface (CLI) that allows the user to completely control VirtualBox from the command line. VBoxManage supports all the features that the GUI gives the user access to, but it supports a lot more than that. VBoxManage must always be used with a specific *subcommand*, such as *createvm, modifyvm, controlvm* and others [6].

The *modifyvm* subcommand allows the user to specify many options for a specific VM, included the networking ones. So, what we have to do is to modify this command so that it supports netmap connector, netmap device name and netmap options. To achieve this we must modify the related source files in order to extend the command.

[6]For a full list of available commands and their functions, refer to [2, chapter 8].

The source files are located in *src/VBox/Frontends/VBoxManage/* and there is a file for each subcommand. Therefore, since we are interested in the modifyvm command, we modified the *VBoxManageModifyVM.cpp*.
We extended the `--nic` option, which specifies the connector, so that it recognizes netmap as an argument. Then, we added two more options to the subcommand:

- `--nicnetmapdev` It specifies the netmap device the adapter will be attached to (just like the *name* field in the GUI.
- `--nicnetmapoption` It specifies the netmap parameters (number of netmap rings and so on).

Furthermore, we modified the file *VBoxManageHelp.cpp*, that is the implementation of the `help` command, so that the above netmap options and extensions are shown to the user.

3.1.3 VirtualBox VM settings

In section 3.1.2 we discussed about how the user can interact with VirtualBox in order to tell it how to configure a VM. In the following, we will show how VirtualBox uses these information to actually configure a VM.

XML settings file

In VirtualBox, each VM has one directory where all files of that machine are stored; in particular, there is an XML file that describes the VM and all its settings.
The settings file contains all those information that a user can specify for that specific VM, in particular the networking ones, using either the GUI frontend or the VBoxManage frontend. Therefore, when a user sets some options for a VM, e.g. specifies a certain connector, the information is stored in the related settings file.

Since we modified the user interface so that it recognizes netmap, now we have to tell VirtualBox how to *parse* the information coming from the UI.

This is achieved by modifying the file */src/VBox/Main/xml/Settings.cpp*, that contains functions which manage the XML settings file. Here we extended the functions that read (write) the settings file so that they are able to return (store) the netmap-related information.

VirtualBox internal configuration

As mentioned in section 2.2.1, when a VM is started, we have three processes: *VBoxSVC* (the server), the frontend process (the client) and the VMM process (that runs the VM).
The VBoxSVC code is splitted in several source files located in *src/VBox/Main/src-server/* directory, due to its modularity, but the only components we are interested in are:

- *VirtualBoxImpl.cpp* It is the main component, the "core" of the server process. It is able to communicate with all other components.
- *MachineImpl.cpp* The *Machine* component. It is an abstraction of a VM. It pares the XML settings file.
- *NetworkAdapterImpl.cpp* The *NetworkAdapter* component. It is the component that manages the networking settings.

The client code is also splitted in several source files located in *src/VBox/Main/src-client/* directory, but the only part we are interested in is the *ConsoleImpl2.cpp* one, since it initializes the VM components.

From now on we will focus only on the handling of the networking settings, since it is the only relevant part for our purposes.

When we launch VirtualBox (e.g. the GUI frontend), the server process starts executing and initializes all the registered VMs through an *init* function. For each VM, that function, asks the machine component to load the (networking) settings for that specific VM from the related XML settings file. After reading the XML file, the machine component gives the NetworkAdapter component the settings information, so that it can store them in its internal structure.

When we launch a VM, before the VM starts executing, the client asks the server process (VBoxSVC) all the (networking) settings details. The information request is then handled by the NetworkAdapter component, that returns the needed information.
The client then:

1. Discriminates which is the port/connector type using the information retrieved from the server.

2. Initializes two configuration data structures that will contain all the networking details returned by the server.

3. Delivers to the VMM the information about which port/connector has been chosen by the user and the related data structure. In that way the VMM knows which port/connector has to instantiate among all registered [7] ones.

So, in the end, when the port and the connector will be instantiated by the VMM, i.e. the VMM will call the respective constructors, the related configuration data structures will be passed to the constructors as arguments.

In order to integrate netmap-related options, we had to extend both server and client codes. In particular we modified the *VirtualBoxImpl.cpp*, *NetworkAdapterImpl.cpp* and *ConsoleImpl2.cpp* source files.

Server code In *NetworkAdapterImpl.cpp* source file we added some statements used to get/set the new netmap options from/in the internal data structure (e.g. when the machine component initializes this network component). Moreover we added two new methods, a *getter* and a *setter*, that are exposed to the main component so that it can get/set the netmap options.
In *VirtualBoxImpl.cpp* source file we added the *getter* method (which calls the new getter function in the NetworkAdapter component) that is exposed to the client so that it can retrieve the new netmap-related information.

[7] See section 3.1.4.

Client code In *ConsoleImpl2.cpp* source file there is a code part that discriminates all the registered connector types. Therefore, we extended that part by adding a possible case in which a netmap connector has been specified. Our code gets the netmap settings[8] from the server, puts them in the configuration data structure and deliver these information to the VMM.

3.1.4 Connector registration

Compiling our new source file and configuring it through a frontend is not sufficient to make the VMM aware of the new connector. In fact, VirtualBox separates the **configuration** from the **registration** of a port/connector.

In the initialization phase the VMM registers all connectors by calling the global function `VBoxDriverRegister` defined in *VBoxDD.cpp* source file, located in *src/VBox/Devices/build* directory. This, in turn, for each available connector in VirtualBox (not necessarily configured), calls the registration function that takes the pointer to the *Driver Registration Record*.
The driver registration record is a data structure containing some information about the connector itself that each connector has to statically allocate. These information are basically the name of the connector, the connector class (in our case is a network class connector) and a list of callbacks, such as the constructor and the destructor callbacks. Since a connector is identified by its name, it must be unique among all connectors. Furthermore, this registration record is useful to the VMM on startup. For these reasons, the structure must have a global scope.

As explained in section 3.1.3, during the startup of a VM, a client tells the VMM which connector has to instantiate and delivers it the related settings information.
The VMM scans the list of registered connectors looking for a connector having a name that matches with the one given by the client. If a match is found, then the VMM instantiates the matching connector and calls the constructor using the pointer specified in its driver registration record.

[8] These settings are the netmap device name (e.g. vale0:1) and, possibly, the netmap options.

It is clear that if we do not register our new connector, the VMM will not be able to instantiate it and, therefore, our code will never be executed.
In order to register our connector, we simply added a couple of lines of code in *VBoxDD.cpp* source file, that call the register function passing the netmap driver registration record as argument. This registration record is contained in our new source file *DrvNetmap.cpp*, which implementation will be described in the next section.

3.2 Connector implementation

In section 3.1 we discussed how we extended VirtualBox so that it is now ”aware” of netmap. In the following, we will show our implementation of connector code.

3.2.1 Internal data structures

As we mentioned in section 3.1.4 we need the driver registration record in order to register our new connector. The most relevant fields in the registration record are:

- Connector name, which identifies it among all other connectors. In our case is ”Netmap”
- The class of the connector. It could be an audio connector, a block connector and so on. In our case it is a network connector.
- Pointer to the constructor function (`pfnConstruct`). It points to our constructor function, which name is `drvNetmapConstruct`.
- Pointer to the destructor function (`pfnDestruct`). It points to our destructor function, which name is `drvNetmapDestruct`.

Once our driver registration record is ready, we must instance the data structure that will implement (and extend) the network connector interface. The data structure contains:

- `INetworkUp` The connector interface data structure. This contains the pointers to the functions [9] that we will implement.
- `pIAboveNet` A pointer to the port interface we are attached to. With this, we can invoke the callbacks [10] exposed to the port.
- `pDrvIns` The pointer to a structure used to retrieve the actual instance of the connector.
- `pszDeviceName` A string containing the name of the netmap device (e.g. vale0:1).
- `pNetmapDesc` The netmap descriptor data structure.
- `pTxSgBuf` A scatter/gather structure (S/G structure). This contains the buffer in which the port writes the frame it wants to transmit and some related information.
- `pIOThread` A pointer to the asynchronous I/O thread.
- `hPipeRead` The read end of a control pipe. It is read only by the asynchronous I/O thread.
- `hPipeWrite, hPipeRead` The write end of a control pipe. It is written by the EMT and by the VMM.
- `XmitLock` The transmit lock that must be acquired by the port.

3.2.2 Connector initialization

Since the constructor is the first called function, this will be our starting point. The function that implements the `pfnConstruct` interface is called `drvNetmapConstruct`. Its purpose is to initialize our internal data structure (see section 3.2.1). It takes the pointer to the structure of the instance of the connector and the pointer to the configuration data structure.
The basic steps this function performs are:

[9] See section 2.2.7, paragraph **Connectors**.
[10] See section 2.2.7, paragraph **Ports**.

1. *Initialization of the connector interface* For each callback exposed to the port, it assigns the related implementation.

2. *Parsing of the configuration* It parses the configuration parameters given by the user [11]. For example, if the netmap device name is missing (which is mandatory), then the constructor returns an error shutting down the VM. The same happens if a user specifies some parameters not supported by netmap.

3. *Opening of netmap descriptor* It calls the `nm_open` function[12] using the user-specified parameters, if any, otherwise it uses the default ones. Opening a netmap descriptor involves the allocation of the netmap rings and, the netmap slots and, therefore, the buffer associated with each slot. Once that these buffer are allocated, their memory is never freed until the netmap descriptor is closed.

4. *Initialization of lock and pipe* It initializes the transmit lock structure and the control pipe.

5. *Thread creation* It instantiates the asynchronous thread, which has to handle the asynchronous I/O operations. The thread is instantiated with a function exposed by the VMM. In that way, the VMM takes care of suspending, resuming and destroying the thread as the VM state changes. In particular, this function takes two functions as arguments: the body of the thread, and the wakeup callback. This is called on the VMM on a VM state change [13].

When the VM is shutdown, the VMM destroys all the connectors instances by calling the destructor (`pfnDestroy`) function on each connector. Our implementation is a function called `drvNetmapDestruct`, and it performs almost all the operations done by the constructor backwards. In particular, it closes the pipe, deletes the lock and closes the netmap descriptor.
The asynchronous thread destruction is performed internally by the VMM.

[11] See section 3.1.3.
[12] Provided by the netmap user API.
[13] See section 3.2.4.

3.2.3 Send side

In this section we explain the implementation of the operations related to the send side. In the following we will show, step by step, what happens on the connector side when a port wants to send a frame [14].

1. `drvNetmapNetworkUp_BeginXmit` Implements: `pfnBeginXmit`. When a port wants to transmit, the first thing it has to do is to acquire the lock on the connector. This function performs a trylock operation [15] on the connector internal lock (`XmitLock`). If this operation is successful, the thread has exclusive access to the connector, so that there cannot be races.

2. `drvNetmapNetworkUp_AllocBuf` Implements: `pfnAllocBuf`. After the lock acquisition, the port asks the connector to allocate the data structure, the S/G buffer structure, where it will store the packet. Since we are using netmap, the buffers associated with the netmap slots are allocated on the initialization phase (constructor), so there is no need to dynamically allocate/free memory. Therefore, we keep a statically allocated S/G structure (`pTxSgBuf`) which parameters will be configured on each `pfnAllocBuf` call. In particular, the buffer pointer in `pTxSgBuf` will point to the available netmap slot buffer. The function returns a pointer to `pTxSgBuf`. This allows to save the overhead caused by memory allocation/deallocation.
   However, it may happen that the netmap TX rings are full, so there is no space available for transmission. If this is the case, we try to flush any potential pending slots by calling a *TX_SYNC* on the netmap File Descriptor (FD) through the `ioctl` system call. If this has no effect [16], we write one byte with value `1` on the write end of the control pipe (the effects will be explained in 3.2.4) and return.

3. `drvNetmapNetworkUp_FreeBuf` Implements: `pfnFreeBuf`. This is called by the port when it wants to free the previously allocated buffer [17]. In our

[14] This implementation is not e1000 dependent, since it must work with any port.
[15] The transmitting thread is the EMT, so it must never block.
[16] E.g. we are transmitting on a netmap pipe and there is no one reading on the other end.
[17] E.g. at the end of transmission, or if an error occurred during some phase.

case, since the buffer points to the netmap slot buffer, this function will simply reset the fields of `pTxSgBuf` instead of actually freeing the memory.

4. `drvNetmapNetworkUp_SendBuf` Implements: `pfnSendBuf`. After the port has successfully filled the buffer, it calls this callback on the connector. This function increments the indexes of the netmap ring and performs a *TX_SYNC* on the netmap FD.

5. `drvNetmapNetworkUp_EndXmit` Implements: `pfnEndXmit`. This simply releases the transmit lock on the connector.

There are two additional callback functions: `pfnSetPromiscuousMode` and `pfnNotifyLinkChanged`. These two functions are not useful for our purposes, but since it is mandatory that a connector implements all the exported methods, we just implemented these two callbacks with two stub functions.

3.2.4 Receive side

In this section we will explain the implementation of the operations related to the receive side. In the following we will show, step by step, what happens on the connector side when a frame is received from the ”outside world” [18].

In section 3.2.2 we stated that the constructor initializes the thread that will handle the asynchronous I/O operations. We called this thread *NETMAP thread*. Mostly, the thread is in charge of handling the incoming frames.

The reception must be necessarily handled by a dedicated thread, and not the EMT, because it has to **wait** for incoming packets. In our case, the thread must do a `poll` system call on the netmap FD [19], which is a blocking operation.
The NETMAP thread body is implemented by `drvNetmapAsynchIOThread` function, while its wakeup callback is implemented by `drvNetmapAsynchIOWakeup`. The wakeup callback writes one byte with value `0` to the write end of the pipe (`hPipeWrite`) and returns. The body of the

[18] This implementation is not e1000 dependent, since it has to work with any port
[19] For further information refer to [1].

thread, instead, performs two different phases: the initialization and the loop.
As a first step, the thread initializes the polling data structures that contain a reference to the FDs to be monitored by the `poll` system call. These FDs are two: the netmap FD and the FD of the read end of the control pipe (`hPipeRead`), both waiting for a POLLIN event.

After the initialization, the function goes in a loop, that iterates until the VM is in *RUNNING* state, that performs the following operations:

1. **Reset of polling data structures**. On the first iteration this has no effect, since those structures have just been initialized. From the second iteration on, since the `poll` modifies the `revents` field of these structures, they must be reinitialized [20].

2. **Wait for a POLLIN event**. It calls the *poll* system call, so that the NETMAP thread blocks until a POLLIN event occurs on one of the FD specified in the initialization phase, or an internal error occurs.

3. **Wake up**. The thread wakes up. This may happen for several reasons, so it must handle all the possible situations:

   - *POLLIN on netmap FD*. ITherefore, it will wait for the port to be ready to receive data by calling the `pfnWaitReceiveAvail` callback on the port itself, passing the `RT_INDEFINITE_WAIT` value as argument; this will cause the thread to block until the port is ready to receive. Of course, if it is not the case, the callback will return immediately.
     When the port is ready, the thread starts a loop in which:
     (a) Fetches the first available frame, if any. Otherwise it exits the loop.
     (b) Checks if the port is able to receive it through the `pfnWaitReceiveAvail` callback. This time, we pass a `0` value as argument, so that if the port cannot receive, the function immediately returns with a negative value. If this is the case, the thread exits the loop.

[20] See the `poll` man page for details.

(c) Pushes it to the port through the `pfnReceive` callback.

(d) Goes back to step (a).

- *POLLIN on control pipe*. It means that the EMT wrote a value in the pipe. If the value is 0, it means that the EMT executed the wakeup callback due to a state change of the VM, e.g. passed from the *RUNNING* state to *SUSPENDING* state. In that case, the thread reiterates the loop, thus it tests the loop condition (VM in *RUNNING* state). The condition will fail this time, since the VM, following the example, is in *SUSPENDED* state. Therefore, the thread it will exit the loop.
  If the read value is 1, it means that the EMT was trying to transmit a frame, but it found all netmap TX rings full (see section 3.2.3). Therefore, the thread, besides waiting for a POLLIN event on the netmap FD, it also waits for a POLLOUT event on the same FD and goes back to step 1. In that way, the NETMAP thread will be awaken even when there is at least one available netmap TX slot.
- *POLLOUT on netmap FD*. It means that at least one netmap TX slot is available for transmission. Therefore, we have to disable POLLOUT event on the netmap FD, and we must notify the port so that it can transmit its pending packets. So we call the `pfnXmitPending` callback of the port and go back to step 1. Notice that this time the NETMAP thread is doing the transmission path.
- *Polling failure*. An error during the `poll` system call occurred. This should never happen, but if does, the thread relinquish the CPU and, whenever it is scheduled, goes back to step 1.

4. **End of loop**. Only happens if the VM is not in *RUNNING* state anymore. In that case the thread function returns.
   As already mentioned, the VMM will take care of re-initializing the thread when the VM will switch back to *RUNNING* state.

Chapter 4

Optimizations on e1000 port and netmap connector

In chapter 3 we described how we extended VirtualBox so that it can make use of netmap framework. In this chapter we will discuss about the problems and bottlenecks related to the current implementation, analyzing the performance. We will prove that our netmap extension is strictly necessary to achieve much better packet rates and, finally, we will propose some optimizations so that we can improve performance even further.

We consider two VMs, we will call them *Guest 1* (G1) and *Guest 2* (G2). The VMs use an e1000 device as a virtual network card, and they are connected through a virtual switch so that they can communicate with each other. Since we want the maximum achievable performance, we installed netmap on both VMs, so that they can use the patched e1000 device drivers and the tools provided by the netmap framework to send/receive traffic at maximum rate.
In particular, guests run *pkt-gen*: a UDP traffic generator that makes use of the netmap framework to speed-up the packet I/O. The application can be run in sender mode and receiver mode. In sender mode, the VM sends UDP packets of a given size at a given rate, while in receiver mode the VM receives all UDP packets it can.

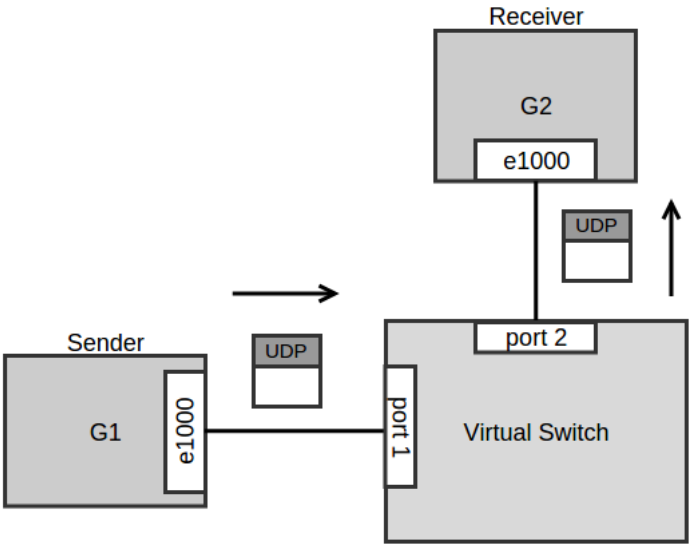

Figure 4.1: An UDP sender running on G1 sends packets to an UDP receiver running on G2. The VMs can communicate with each other through a virtual switch.

In our case G1 will act as a sender and G2 as a receiver [1]. This situation is shown in figure 4.1.
Starting from this situation, we will analyze two scenarios: in the first case the VMs will be attached to the virtual switch provided by *VBoxNetAdp* kernel module (section 2.2.2) called *vboxnet0*. So we will use the host-only networking mode (see section 2.2.6). In the second case the VMs will be attached to a VALE switch, in particular G1 will be connected to the first port (*vale0:1*) and G2 will be connected to the second port of the switch (*vale0:2*).

The first scenario will be our reference for performance analysis. In this way we will prove that the second scenario is the most efficient in terms of packet rate.

[1] Since both VMs run the same OS (Ubuntu 15.10), we can swap the roles of G1 and G2 without any issue.

4.1 Analysis of current implementation

Now we are going to discuss about the performance of the current implementation (see section 2.4 [2]) using both host-only and netmap networking modes.

In this section we will make two experiments, both with G1 acting as a sender and G2 acting as a receiver. In the first experiment we will run a simple application that will send (receive) UDP packets using a simple UDP socket in an infinite loop. This will be our first reference in terms of performance. In the second experiment, instead, we will run the aforementioned pkt-gen application that will make use of the netmap framework. This application is just an infinite loop that, for each iteration, fills its local netmap TX ring with a batch of packet (256) and performs a TX operation. In both cases, the packets are 60 bytes long.

4.1.1 Using host-only networking mode

We first analyze the case where the VMs are connected with each other using the host-only networking mode, in particular using the *vboxnet0* virtual switch provided by VirtualBox. We are going to analyze TX and RX performances separately.

TX performance

The measurements are shown in table 4.1. All the values are computed counting the number of occurrences of each event over a period of 1 second and dividing the count itself for that time period.

When we use a UDP socket, we have a TX packet rate that is not really high, about 30.5 Kpps. We can see that there is a TX notification (i.e. TDT write) and an interrupt for each packet. *TX descriptor load* is the number of times the emulation code accesses the guest physical memory to read the transmit descriptors, while *TX descriptor fetched* is the average value of fetched TX descriptors in a single memory access.

[2] For now we keep both TX and RX descriptor caches enabled.

Measured quantity	UDP Socket	Netmap native mode
TX packet rate	30.5 KHz	340 Kpps
Interrupt rate	30.5 KHz	1.35 KHz
TX notifications	30.5 KHz	1.35 KHz
TX descriptors load	30.5 KHz	6.8 KHz
TX descriptor fetched (average)	1	51

Table 4.1: This table shows statistics about the sender VM attached to the *vboxnet0* virtual switch. The *descriptor load* shows how many times the guest physical memory is accessed to read the TX descriptors.

Even if we enabled the cache for transmit descriptors, we are not effectively taking advantage of it (only one fetched descriptor per memory access). This is due to the fact that we have one TX notifications per packet. When the device driver in the guest writes in the TDT register, the EMT performs a VM exit and executes the emulation code. As a result, when it wants to transmit, it finds only one descriptor that must be processed. After the descriptor processing, the EMT performs a VM entry and executes the device driver, that, in turn, will prepare another descriptor and write in TDT register and the EMT will go back to the emulation code in the host world. This procedure introduces a lot of overhead.

The problem of TX notification is not present when we run pkt-gen, since it exploits the modifications provided in the patched e1000 device driver. In fact, for each TXSYNC operation invoked by the application, the driver pushes down packets in batch, resulting in one TDT register write per batch instead of one TDT write per packet. As a result, we have an improvement of performance that is about 10 times than before (about 340 Kpps).

Measured quantity	UDP Socket	Netmap native mode
RX packet rate	30.5 Kpps	340 Kpps
Interrupt rate	14.8 KHz	31.8 KHz
RX notifications	14.4 KHz	2.11 KHz
RX descriptors load	2 KHz	23.08 KHz
RX descriptor fetched (average)	15	15

Table 4.2: This table shows some statistics about the receiver VM attached to the *vboxnet0* virtual switch. In this case the RX descriptor cache is exploited.

RX performance

All measurements are shown in table 4.1. Again, the values are computed counting the number of occurrences of each event over a period of 1 second and dividing the count itself for that time period.

When we use UDP Sockets the packet rate is the same as the transmitter, so it is able to receive all the packets coming from the switch.
We can see that the RX descriptor cache is exploited this time. This is due to the fact that the cached RX descriptors are the free descriptors that the adapter can fill with incoming packets (see section 2.4.2), so it is not strictly dependent on the RDT register writes performed by the device driver on the guest.

When we run pkt-gen, we can see an improvement of performance in terms of packet rate. In fact, we can see that the RX notification rate decreased greatly decreased even if the packet rate is about 10 time higher than the previous case. This is thanks to the fact that the modified e1000 driver writes to the RDT register only at the end of the interrupt routine, not for each packet.
We can also notice that the interrupt rate is much higher (31.8 KHz) with respect to the sender one (1.35 KHz). The reason is that, potentially, in the RX case an interrupt is raised for each received packet. This is unavoidable because the

Measured quantity	UDP Socket	Netmap native mode
TX packet rate	23.8 Kpps	1.148 Mpps
Interrupt rate	23.8 KHz	3.8 KHz
TX notifications	23.8 KHz	4.5 KHz
TX descriptors load	23.8 KHz	22.4 KHz
TX descriptor fetched (average)	1	51
TXSYNC rate	26.3 KHz	1.148 MHz

Table 4.3: This table shows some statistics about the sender VM attached to the *vale0* virtual switch. In native mode we have a great improvement of performance in terms of packet rate.

adapter cannot guess whether or not an incoming frame is the last of a batch.

4.1.2 Using netmap networking mode

Now we analyze the case where the VMs are connected with each other using the netmap networking mode, in particular using the VALE virtual switch provided by netmap.

TX performance

The measurements are shown in table 4.3.
Using UDP packets, we have a packet rate that is even lower than the host-only case. This is due to the fact that, since the netmap connector is designed for packet batching, the overhead is not amortized over a batch of packets, so we have a decrease of performance.

In netmap native mode, instead, we can see that the performance dramatically increased to 1.148 Mpps, that is **over 3 times better** than the host-only case with the adapter in native mode.

Measured quantity	UDP Sockets	Netmap native mode
RX packet rate	23.8 Kpps	588 Kpps
Interrupt rate	15 KHz	23.2 KHz
RX notifications	15 KHz	5.4 KHz
RX descriptors load	1.7 KHz	42.7 KHz
RX descriptor fetched (average)	14	14

Table 4.4: This table shows some statistics about the receiver VM attached to the *vale0* virtual switch. Also here, in native mode we have a great improvement of performance in terms of packet rate.

The TXSYNC rate shows how many times the TXSYNC operation is called in the connector. As we can see, we have one TXSYNC per frame. This means that, even if the e1000 adapter receives batches of packets, they are sent on the VALE switch one by one. Since the connector knows nothing about the packets arriving from the port, this situation is unavoidable, unless we do not modify the e1000 emulation code.
Ultimately, if we were able to perform only one TXSYNC per batch, we would see a great improvement of performance. A solution will be presented in section 4.2.

RX performance

The measurements are shown in table 4.4.
As far as concerns UDP sockets experiment, all the discussions made for the previous cases are still valid.
In netmap native mode, we can see a quite improvement of performance, about 3/4 times faster than the generic case, about 73% than host-only case with the adapter in native mode. The reason why the performance are not as high as the transmission ones, resides in the interrupt rate. Since the adapter knows nothing about packet batches, it raises an interrupt as explained in section 2.3.3. Even if

the connector is able to understand whether or not the current packet is the last one of the batches, there is no way to send this information to the port without modifying the port itself. Therefore, in order to improve performance, we must modify the e1000 emulation code. These modifications will be explained in section 4.2.

4.2 Packet batching

In the previous section we discussed about the performances using both host-only networking mode and and VALE switches, and we proved that we achieved very high performance using the latter.
Moreover, we showed that we can further increase both TX and RX performances by making the connector and the port aware of packet batching. In this section we will present our solution.

4.2.1 Implementation

In order to achieve higher performance on both TX and RX sides, we made some modifications on the e1000 port and on our netmap connector.

TX path

On the TX path, we modified both port and connector. In particular:

- We added some lines of code where the port calls the *pfnSendBuf* callback on the connector. Before calling it, we check if the current outgoing frame is the last one in the batch by looking at TDT and TDH registers.
  Let N_s be the number of TX descriptors. If

  $$(TDT - TDH) \bmod N_s \equiv 1$$

  then we set a flag (*TXSYNC* flag) on the S/G data structure passed to the callback as an argument, so that the connector understands that this is the last frame in the batch.

- In the implementation of the *pfnSendBuf* callback in the connector, we check if the *TXSYNC* flag is set. If this is the case, then we perform the TXSYNC operation, otherwise we just update the netmap indexes and return.

In that way, instead of doing a TXSYNC for each outgoing packet, we perform that operation only at the end of the batch.
We must point out that we did not add anything new in the S/G data structure. In fact, this structure contains a string of 32 bits used for flags. 28 bits of this string are used for other purposes, while the remaining four bits are available for new custom features.

The modification on the port has effect only in netmap networking mode, since the flag we set are checked only by our connector.

RX path

Also on the RX path, we modified both port and connector. In particular:

- In the asynchronous thread body, we enter a loop in which we call the *pfnReceive* callback on the port until there are available incoming frames or until the port is not available for receiving anymore. After the loop, we added a further call to the *pfnReceive* callback, passing a *NULL* pointer instead of available data. This is our way to notify the port that there are no more packets in the batch.

- We added a check on the *pfnReceive* implementation of the e1000 port, in which we check the pointer to available data passed as argument. If it is *NULL*, it means that the batch is finished, so we raise an interrupt and return.
  Moreover, since we want to raise an interrupt only in that case, we disabled all other interrupts in the receiving path.

In this case the modifications we made are compatible **only** using the netmap networking mode and the e1000 device as emulated network adapter. In fact, the modified port will raise an interrupt **if and only if** it has a *NULL* pointer as

Measured quantity	**TX stats.**	**RX stats.**
Packet rate	1.615 Mpps	940 Kpps
Interrupt rate	6.3 KHz	**2 KHz**
TX/RX notifications	6.3 KHz	83 KHz
Descriptors load	32 KHz	63.2 KHz
Descriptor fetched (average)	51	15
TXSYNC	**6.38 Khz**	/

Table 4.5: This table shows statistics about both VMs attached to the *vale0* virtual switch. We highlighted the items that caused the performances improvements.

argument, but only our connector will pass this value to the port. Therefore, if we change the networking mode (e.g. host-only), the guest will not be able to receive any packets.
The same *may* happen if we change the port. Since it is not suppose to happen that a connector passes a *NULL* value to the port instead of the available data, the effects of this particular function call may be unpredicted and may also cause a crash of the VM itself.

4.2.2 Performance analysis

The results are shown in table 4.5. As we can see, we have a great improvement of performances. We achieved 1.615 Mpps on the sender and 940 Kpps on the receiver.
As highlighted in the table, we can see that the TXSYNCs are performed only at the end of the batch. Also the interrupt raised to the guest from the port are now amortized over the batch.

We can also notice that the receive rate is less than the transmit rate. That is caused by a bottleneck in a chained producer-consumer system.

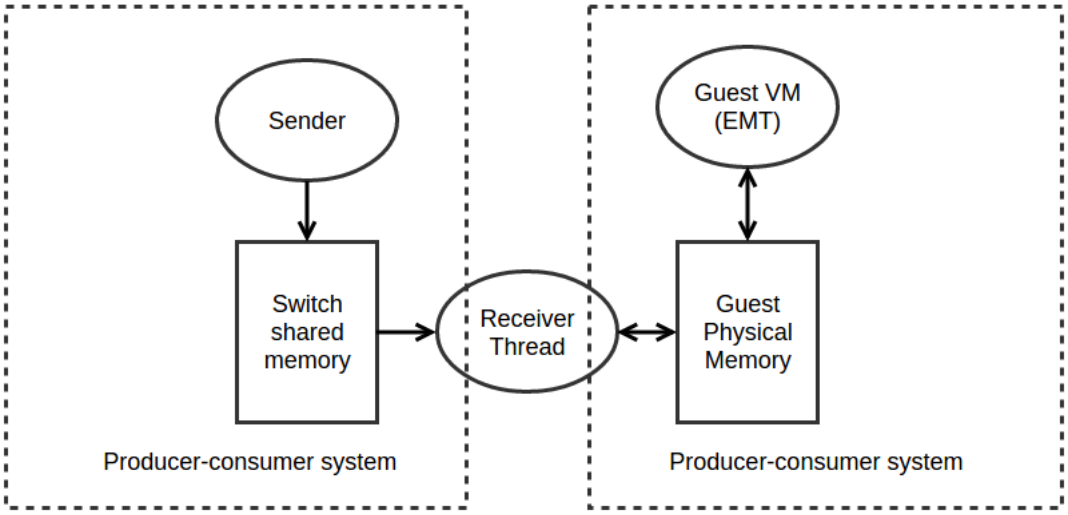

Figure 4.2: Two producer-consumer systems forming a chain. The first system is composed by the sender EMT as a producer and the Asynchronous I/O thread on the receive side as a consumer. The second system is composed by the the same asynchronous thread as a producer and the receiver EMT as a consumer.

Figure 4.2 depicts our situation. We have the sender EMT that sends frame using the virtual switch. The receiver thread reads the new available data, writes them in the guest physical memory, and raises an interrupt to the guest so that the EMT of the receiver VM can consume the new data.
In this system, the slow part is on the receive side. The reasons for this slowness are multiple, such as synchronization between the receiver thread and the receiver EMT and context switches between guest world and host world. The overhead introduced by these factors is not manageable since it is strictly dependent on the computation power of the host.

4.3 Implementing mapping of descriptors

Looking at the previous tables, in particular table 4.5, we can see the *Descriptor load* row, which has a value of 32 KHz for transmitter and 63.2 KHz for receiver. As already mentioned, this value measures the accesses in guest physical memory performed by the emulated adapter. Each guest physical memory access involves an address translation, that implies some overhead. At *normal* packet rates (e.g.

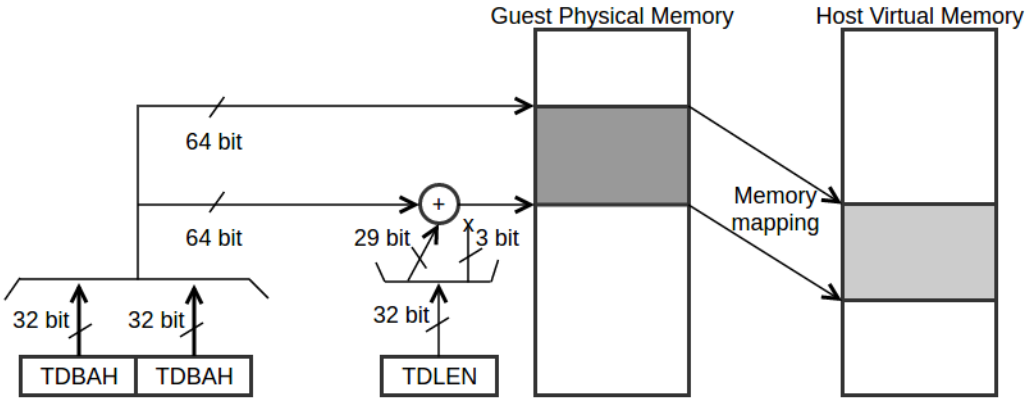

Figure 4.3: The mapped memory region for TX descriptors. The RX descriptor are mapped in the same way, but with the RX related registers. Notice that TDLEN is right-shifted by 3. This represent the division by 8.

about 30 Kpps), this overhead is negligible, but when we deal with very high packet rates, this becomes notable.
Therefore, we adopted a solution that avoids those translations by *mapping* a portion of the **guest physical** memory into the **host virtual** memory, so that the descriptors can be accessed by the host world using pointers.

4.3.1 Memory region containing descriptors

In order to map the e1000 descriptors in the host virtual memory, we need to know where the descriptors are located in the guest physical memory. To do so, we look at the aforementioned *TDBAH* and *TDBAL* registers for TX descriptors, and *RDBAH* and *RDBAL* registers for RX descriptors. In this way, we find the base address of the physical pages containing the descriptors.
Moreover, the *TDLEN* and *RDLEN* registers contain the length in bytes of the physical memory occupied by the TX descriptors and RX descriptors, respectively.

Since the descriptors must be accessed in DMA by the adapter, we have the guarantee that the descriptors are allocated in contiguous pages [3] in physical

[3] In most Operating Systems, the size of a page is 4 Kilobytes.

memory.
Therefore, the mapped region will start from the base address

$$TDBAH|TDBAL$$

up to

$$TDBAH|TDBAL + TDLEN/8$$

address for the TX descriptors [4].
Figure 4.3 depicts the memory region mapped in the host virtual memory.

Since each descriptor is 16 bytes long, and a memory page is 4 kilobytes, in a single page we can map up to 256 descriptors, which is the default number of descriptors per ring that the device driver sets by default. Furthermore, the e1000 devices support up to 4096 descriptors per ring, so our implementation supports the mapping of up to 16 memory pages, containing 256 descriptor each.

4.3.2 Implementation

Since we want to map the descriptors, we do not need any descriptor cache. Therefore, we disabled both TX and RX descriptor caches on the e1000 emulation code.

We implemented the mapping of the region using a function provided by VirtualBox: *PDMDevHlpPhysGCPhys2CCPtr*. This function maps a guest physical memory page into the virtual memory of the host. It takes, among the others, the base address of the physical memory and a pointer that, after the execution of the function, will be used to access the mapped page.
Since the base address of the descriptors may change during the execution of the VM (i.e. the value TDBAH/RDBAH and TDBAL/RDBAL registers), or even the number of descriptors itself (i.e. TDLEN/RDLEN), we customized the callback functions for write operation of these registers so that we can remap the pages when the registers values change.
The mapping function is only available in R3 context, but the callbacks are

[4] The mapping of RX descriptors is analogous.

always called in R0 context. Therefore, we added two new event queues to the port, one for TX descriptors, one for RX descriptors (see section 2.4). When there is a write in one of the descriptors-related registers, the EMT:

1. Stores the old value of the register.
2. Updates the register.
3. Since it is in R0 context, posts an item in the event queue, so that it will execute the related callback after it switches to R3 context, but before it resumes executing guest code.
4. Executes the related callback function. This function checks whether or not the new value is equal to the old value. If it is true, then simply returns, otherwise it unmaps all the mapped memory pages (if any) and remaps them back using the new value of the register.

All these operations may be expensive in terms of execution time, but since a remapping operation seldom happens (e.g. the reset of the adapter), its cost is well amortized.

This new optimization is completely compatible with other networking modes, since it does not affect the interaction between port and connector.

4.3.3 Performance analysis

Thanks to this optimization, we do not need to translate the guest physical address each time we need a descriptor. Instead, we can directly access it by means of a pointer.
Table 4.6 shows the performances in both TX and RX sides.

We can see that we have an improvement of performance in terms of packet rates that is 33% in TX case and 45% in RX case.
Since we disabled descriptors caches, the adapter would have loaded only one descriptor per memory access. Therefore, thanks to memory mapping, we saved

Measured quantity	TX stats.	RX stats.
Packet rate	2.2 Mpps	1.370 Mpps
Interrupt rate	8.9 KHz	2.8 KHz
Descriptors accesses	2.2 MHz	1.37 MHz

Table 4.6: This table shows the performances when we enable both packet batching and memory mapping. Since we disabled descriptors caches, we have an access for each packet transmitted/receive

the overhead time that a total of **3.5 millions** of address translations would have caused for each packet.

4.4 Code optimization

We can further improve performance by means of small optimizations in the e1000 port.
The emulation code contains some statements that are not useful (e.g. collecting statistics) and, at very high packet rates, introduce a notable overhead because they are executed for each single packet. We deleted those statements to improve performance:

- *Useless lock acquisitions* Acquiring a lock may cause overhead. In the code there are some locks that are useless: as a matter of fact, also the documentation contained in the code itself says that one of these locks is useless. Therefore, we eliminated these locks.

- *Collecting statistics* In the code there are some functions whose purpose is to collect some statistics. These functions introduce overhead since they need to acquire a lock (that we already deleted) but also they perform some memory comparisons that introduce some overhead. Therefore, we deleted the calls to those functions.

- *Extra memory copy* In the RX path, for each received packet, there is copy

Measured quantity	TX stats.	RX stats.
Packet rate	2.2 Mpps	1.750 Mpps

Table 4.7: This table shows the packet rates after our code optimizations and when we enable both packet batching and memory mapping. We have an improvement only on RX side because it is the one that mostly depends on the computation power of the host.

from a local buffer to another local buffer, which is completely useless. Therefore, we avoid this extra copy.

As already mentioned, these statements would not affect performance with normal rates, but since we are dealing with millions of packets per second, the effect of a negligible overhead is amplified hundreds of times, so that it becomes notable.
In table 4.7 the packet rates after these optimizations are shown.

As we can see, we have an improvement in terms of rate of received packets, while the TX packet rate remains equal to the previous case. This is due to the fact that the optimizations mostly concern the RX path, and, moreover, this is is the part that is strictly related to the computation power of the host. Therefore, it is easy to understand that avoiding statements that imply some expense in terms of computation time, it can quite improve performance.

Chapter 5

Conclusions

In this work we showed the architecture of VirtualBox, in particular we gave details on the virtualization and networking architecture parts. We then presented our extension to VirtualBox so that it can support the netmap framework, implementing a new connector that is able to connect a VM to a VALE virtual switch. Subsequently, we analyzed the performance of the existing implementation using an e1000 device as virtual adapter and comparing the performances with the host-only networking mode and our new extension, pointing out the bottlenecks and problems. We also presented solutions to these problems exploiting packet batching, memory mapping and optimization of the e1000 emulation code in order to further improve performance.

We successfully improved performance starting from 340 Kpps in the host-only case, both TX and RX rates, achieving up to 2.2 Mpps in transmission and 1.75 Mpps in reception using netmap mode with optimizations. This means that we improved the TX packet rate of **6.5 times** and the RX packet rate of **5 times** with respect to host-only mode.

Bibliography

[1] Rizzo L. *netmap: a novel framework for fast packet I/O*

[2] VirtualBox user manual. https://www.virtualbox.org/manual/ http://info.iet.unipi.it/~luigi/papers/20120503-netmap-atc12.pdf

[3] Rizzo L., Lettieri G. *VALE: a switched ethernet for virtual machines.* http://info.iet.unipi.it/~luigi/papers/20121026-vale.pdf

[4] AMD *Secure Virtual Machine Architecture Reference Manual.*

[5] Neiger, Gil Santoni, A. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10, 3 (2006)*

[6] Yehuda, B. Utilizing IOMMUs for virtualization in linux and xen.

[7] Agensen, O., Mattson, J., Rugina, R., Sheldon, J. Software techniques for avoiding hardware virtualization exits.

[8] Russel, R. virtio: towards a de-facto standard for virtual I/O devices.

[9] Intel *Intel 64 and IA-32 Architectures Software Developer's Manual.*

[10] Intel *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developers Manual.*

[11] Qt project. http://www.qt.io/

[12] SDL project. http://www.libsdl.org/index.php