Universitá di Pisa
Facoltá di Ingegneria
Laurea magistrale in Ingegneria Informatica

Tesi di Laurea

Optimizations in Virtual Machine Networking

Relatore

Prof. *Luigi Rizzo*
Universitá di Pisa

Candidato

Vincenzo Maffione

Correlatore

Prof. *Giuseppe Lettieri*
Universitá di Pisa

Anno Accademico 2011-2012

Abstract

Network performance is a critical aspect in Virtual Machine systems and its importance is becoming increasingly important in the world of computing. These systems are commonly employed in the IT departments of several organizations, since they allow IT administrator to build network services with high reliability, availability and security, or to improve efficiency in computing resource usage.

In this thesis we are going to analize the state of the art of virtual machine networking, evaluating advantages and drawbacks of the existing solutions. We then propose a new approach, showing that with a small amount of code modifications, we can bring a classic emulated network device (we take `e1000` as a reference example) to a performance that is comparable or superior to the performance of paravirtualized solutions.

Contents

Chapter 1

Introduction

Standard computer systems are hierarchically organized in a three layers stack, as depicted in figure 1.1. The lowest layer is the bare hardware, the middle one is the operating system and the upper layer contains the applications software. Two neighbor layers can communicate through a well-defined *interface*, so that each layer can ignore how the lower layers are actually implemented. In this way, the interface provides an *abstraction* of the underlying software/hardware resources.
This (relatively) simple architecture has proven to be very effective in delivering the IT services required by companies, individual users and other organizations.

Neverthless, more complex computer systems organizations have been devised and used, in order to overcome some limitations of the standard computer system model.
These computing environments are are known as *Virtual Machines* (VMs) systems. By itself, the term *Virtual Machine* can have several meanings, so when using it is important to point out what we are addressing. In section 1.1 we will provide a classification of these meanings. In general *virtualization* provides a way of increasing the *flexibility* of real hardware/software resources. When a physical resource is virtualized, it can appear to be a resource of a different kind or even a *set* of different resources (the same kind or different kind).
As an example, a single IA32 processor can be virtualized in a way that emulates more PowerPC processors. Once this is done, you can build many standard 3-layer computer systems on top of each emulated PowerPC (virtual) processor, using unmodified OS and applications designed to be used on the PowerPC architecture.

We talk about Virtual Machines when we virtualize a physical computing enviroment (e.g a server machine) and get one or more independent virtual computing environment (the VMs), potentially different by the original one.
In the VMs terminology, each VM is also called *guest*, whereas the virtualized physical computing environment is known as *host*. The piece of software that provides the support for virtualization is called *Virtual Machine Monitor* (VMM)

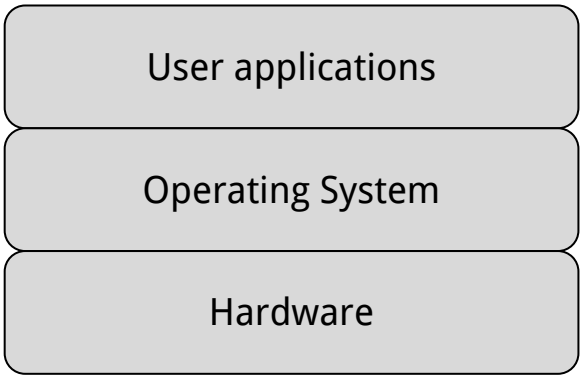

Figure 1.1: A standard 3-layers-stack computer system. On the bottom the bare hardware, in the middle the Operating System, on the top the Applications layer.

or *Hypervisor*.
You can virtualize nearly all the resources you wants: disks, network devices, memories or other peripherals.

Generally speaking VMs allow to build computer systems with more abstraction levels than the standard model has. This has important advantages:

- In terms of *flexibility*, using VMs you can easily run programs compiled for a given Instruction Set Architecture (ISA) and a given Operating System (OS) on top of a computer system that has a different ISA and/or a different OS. Using a standard system you would be bound to the ISA of your processor and the operating system installed on your machine. This flexibility can be exploited in several situations, such as testing new software on different architectures (without physically have the machines supporting each different architecture), or run legacy applications on newer, more power-efficient hardware.

- In terms of *protection*, VMs can provide multiple isolated execution environments running on the same phisical machine. This allows to execute different applications in different VMs (each VM can have its own OS), so that if an application has a security hole, an attacker cannot use the hole to do malicious attacks to an applications running on a different VM. This scenary is still possible when applications are run in the same OS.

- In terms of resources usage, VMs can help to reduce hardware costs and power consumption, since they naturally improve resource utiliziation. For instance, you can use only one physical server machine to provide multiple services (without sacrifying isolation), using the 100% of the machine re-

source, instead of using many underutilized server machines [1]. This results in money and energy saving.

- In terms of *mobility*, you can easily migrate VMs (and so replicate them) to other locations, simply transmitting some files through the Internet. This can also help avoiding setup times (software installation and configuration), since through a VM you can convey a ready-to-use copy of a computing environment to the user.

The previous list is not exhaustive, but gives an idea of the services that virtualization can deliver, and makes clear the reasons why IT departments make massive use of virtualization technologies.

1.1 Virtual Machines classification

As noted previously, the term *Virtual Machine* can have several meanings. Therefore it is useful to give a classification of the possible meanings (see [9] section 1.5 in [14]).
First of all, VM can be divided in two categories:

- *System Virtual Machines*: these VMs provides virtualization at ISA level. This means that the VM is capable of executing arbitrary code compiled for a specified ISA. System virtual machines provides a complete executing environment where multiple processes can be run. A system VM can then be used to run an OS that supports several applications, namely a standard 3-layers computer environment.

- *Process Virtual Machines*: these VMs virtualize at the Application Binary Interface (ABI) level, providing an execution environment for a single application. Since applications are usually written in high level languages and so use an high level interface[2], if we want to execute a single application, the VM is only required to emulate the high level interface and/or a subset of the ISA[3]. User applications are therefore provided with a virtual ABI anvironment.

1.1.1 System level Virtual Machines

System virtual machines can be further divided depending on whether the code executed in the VM (the guest) is of the same ISA of the physical machine supporting the VM (host).

[1] Assuming, as often happens, that one or a few services don't utilize all the computing resource offered by a modern server machine.

[2] For instance an OS system call interface, or the interface provided by an interpreted programming language.

[3] Typically unprevileged instructions, and instructions that are not problematic with respect to CPU virtualization (see [2] and section 8.2 in [14]).

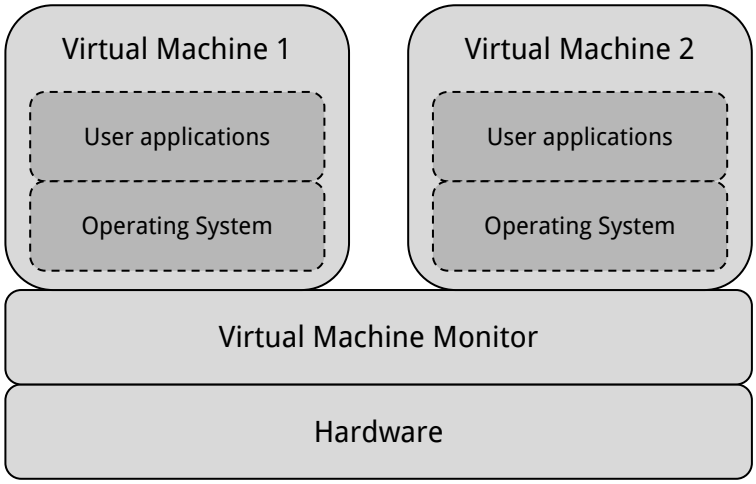

Figure 1.2: An example of system using Type 1 VMMs. The VMM runs on the bare hardware.

The same-ISA case is very common, since users are often interested in server consolidation (resource usage optimization), protection or live migration, but don't care about executing code compiled for a specific ISA. Same-ISA VM are generally easier to design and are generally more suitable to be executed efficiently (e.g. the efficient hardware-based virtualization can be used in this case).
Same-ISA system VM can be further divided, depending on how the VMM is implemented:

- Type 1 VMM (*Native* VMM). In this case the VMM is a software component that runs on the physical machine (the host) without any OS support. It's up to the VMM to interface directly with the physical resources of the server machines, such as CPUs, memory and peripherals. Type 1 VMM can deliver a very good performance, but are more complex to design and require more development efforts, since there is no OS providing basic services, abstractions, device drivers and the like. An example of system including a Type 1 VMM is illustrated in figure 1.2.
- Type 2 VMM (*Hosted* VMM). In this case the VMM it's just a regular OS process, that runs on the host OS along with other processes. The VMM can access the physical resources of the host machine through the OS services. The OS support speeds up the development process and make the VMM portable. On the other hand, performance is generally inferior with respect to the Type 1 VMM. An example of system including a Type 2 VMM is illustrated in figure 1.3.

The different-ISA case is useful if you want to run legacy software on modern

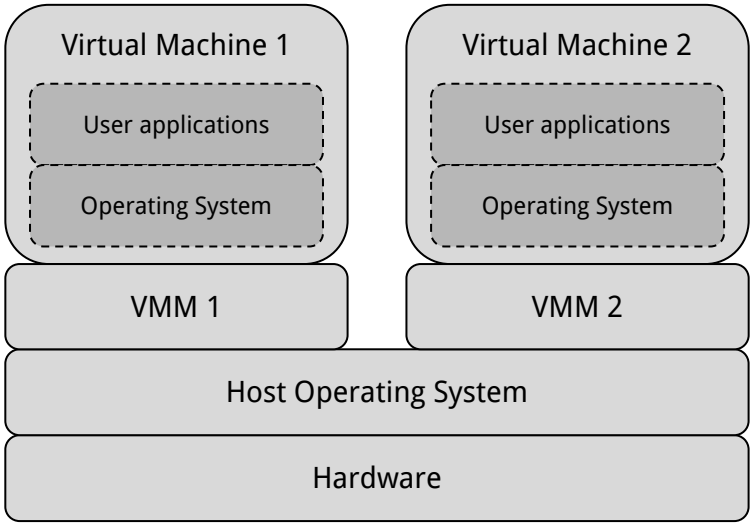

Figure 1.3: An example of system using Type 2 VMMs. The VMM runs on top of a conventional OS.

machines[4] or if you want to test your software for compatibility with other architectures. In this case a VMM is basically a full system emulator, capable to emulate the complete behaviour af a complex computer system.

1.1.2 Process level Virtual Machines

A very similar secondary classification applies also to process virtual machines, although at the ABI level. A VM can expose to the application it executes (the guest application) the same ABI exposed to a regular process excecuting in the host system, or expose a completely different ABI.
Here the common case is when the ABI is different. The Java Virtual Machine is an example of this VM type: it executes code written conforming to an ABI (the Java bytecode) which is different by the ABI offered by the host OS. In this case the JVM engine is the VMM. All interpreted language engines are VMs of this kind. Process VMs are also used for other purposes, such as runtime optimization of binary code (see chapter 4 of [14]), or binary translation in general.

The same-ABI case is just the concept of multiprogrammed OS, namely an OS capable of virtualize the CPU and memory, offering to each process a *virtual processor* and a *virtual memory*. These two virtual resources make up the enviroment (which provides some limited degree of isolation) where an OS process lives. We are so used to the multiprogramming concept that we don't even think of it as being a form of virtualization. In this case the VMM is the OS itself (together with the

[4]Game console emulators are a common case of this kind of VM.

ISA which is designed to efficiently support multiprogramming).

1.2 Virtual Machine Implementation

As showed in section 1.1, there are many types of conceptually different virtual machines. Neverthless, all the VMs deal with executing code written for a certain environment (source code, or guest code), using another environment (the host environment). This is a form *emulation*: the term emulation refers to *the process of implementing the interface and functionality of a system on a system having a different interface and functionality* ([14] page 27).
The basic techniques employed to implement emulation are three: interpretation, dynamic translation, and hardware-based virtualization. These techniques can be used alone or in combination to provide the virtual environment we want to implement. Nowadays VMMs generally use a combination of the three methods.

1.2.1 Interpretation

The naive emulation technique is known as *interpretation*. Basically, the VMM has to do in software what a physical CPU would have done in hardware. Take the current instruction (or statement, if we are dealing with high level languages VMs), excecute it updating the VM status (which is an in-memory representation of all the VM resources, like registers, memories and the like), and go to the next instruction. The VMM would then be implemented as a loop that, in each iteration, performs the fetch, decode and execute phases of instruction execution.

Although writing an interpreter for a modern ISA or an interpreted programming languages can be a very long and complex process, because of the complexity of the source language, the technique is conceptually easy. You just have to read an Instruction Set specification or a Programming Language specification and implement all the possible instructions/statement strictly respecting the specified behaviour.
Being simple, this method is generally terribly inefficient if compared to the native execution of the source code on a processor designed to execute the source ISA, because for each source instruction the VMM has to execute many host instructions (e.g. 30-100) to perform in software all the necessary operations. In other words, the average *translation ratio* is very high (e.g. 40). On the other end, a big advantage is that the VMM has always the control over the execution of the source program, because it is executing the program step by step.
Many tricks can be implemented in order to improve the emulation performance. Some of these tecniques can be found in [14] (sections 2.1 through 2.4).

1.2.2 Dynamic Translation

A more sophisiticated form of emulation is called *dynamic translation* or *binary translation* or *Just In Time compliation*, depending on the context.
Having to translate a source code in something else, an idea is to translate it into equivalent binary code that can be directly executed on the host CPU. The VMM does this translation *on the fly*, as soon as the source code is fetched from memory. The method is intended to amortize the costs of interpretation, doing the repetitive work (fetch and decode) once or a few times. The code execution step of a source instruction or a block of instructions, namely the translation, is generated once (or a few times) and saved in a *Code Cache*. The next time the source program flow goes through that block of source instructions, we don't have to fetch, decode or translate, but just to execute the translation block. The blocks of translated instructions can be connected directly to each other using native jump instructions, in the way suggested by the source program flow. After some time the code cache will become the complete translation of the source program into the host ISA.
The final result is that the average translation ratio can be very close to 1 (e.g. less than 4), giving a nearly native performance, or at least a performance that is acceptable also for performance sensitive applications.

The whole process is of course way more complicated than what has been presented here. Several problems are present, such as the *code-discovery* problem that makes impossible to do a static translation, or the *code-location* problems, that is due to the different address space of the guest and host systems, or the state mapping problem, that is the way the VMM maps guest registers and similar resources to the host ones.
Similarly to the interpretation method, with dynamic translation the VMM has (or can easily get) complete control over the guest code execution: While doing the translation, it can put *traps*[5] in the guest code wherever it wants.
For further informations on dynamic translation, see [14] (sections 2.5 through 2.9).

The same-ISA case

Interesting enough, interpretation and dynamic translation can make sense also in the same-ISA case. In this case the translation is simplified, and most of the time the source code can execute natively on the host machine, without performance losses.
However there are some instructions that cannot be executed natively, because they access physical resources, because are trying to access resources that do not exists on the physical machines, or because they are not easily virtualizable (see [2] and section 8.2 of [14]).

[5] Point in the code that interrupts the guest execution and give control to the VMM directly, or indirectly thorugh the host OS.

As a typical example, memory accesses addressing the I/O space or the memory mapped I/O can have side effects and then must be emulated in software. If the instruction was intended to access a physical resource that exists on the host, like a network adapter, the VMM cannot allow direct access to the device, because other processes or the host OS could be accessing the same device at the same time, and certainly the host network driver and the guest network driver are not aware of each other. If the instruction was intended to access a virtual network adapter (that doesn't exist on the host), the I/O instruction must be trapped in order to emulate the device behaviour in software.

1.2.3 Hardware-based virtualization

Due to the widespread use of VMs, the processor vendors have introduced processor extensions that allow for efficient and safe execution of guest code in the same-ISA case. These hardware assists are intended to overcome some of the common problems arising when using dynamic translation techniques, and at the same time they make it easy to execute guest code natively. For the x86 ISA, both AMD and Intel have proposed their extensions, AMD-v ([4]) and Intel VT-x ([10]). These features provide all the means necessary to fully virtualize the x86 ISA. Since they fairly complex, we will only outline those aspects that are intersting with respect to our work.

When the extension are present, the CPU is able to switch to a special mode, that we will call *VM mode*, through a so called *VMEnter* instruction and switch back to normal mode through a so called *VMExit* instruction. When in VM mode, the CPU can execute guest code in a controlled environment. When necessary, the CPU can switch back to normal mode, starting to execute host code (VMM, OS or other processes). The switch operation between the the host world and the guest world is conceptually similar to the more familiar process context switch, since it includes saving the host (guest) state and loading the guest (host) state. These operation are done in hardware but are still very expensive, expecially if we consider the additional software overhead involved in this host-guest transition, due to OS operations and possible userspace/kernelspace transitions that could be necessary to transfer the control to the VMM or to the guest.
The VM switches are necessary in some situations, such as dealing with I/O operations (see 1.2.2), or when we want to deliver an interrupt to the guest.
Since VM switches are very expensive, but sometimes necessary, trying to minimize them is fundamental if a VMM wants to deliver good I/O performance.

1.3 I/O Virtualization techniques

Similarly to what happens with code interpretation (section 1.2.1), emulating a device means doing in software what the device would do in hardware. Therefore,

each time the guest accesses an I/O device (e.g. writes to a device register), the VMM has to take the control and emulate all the *side effects* associated to the specific I/O access.
Virtualization of I/O devices is also described in [15], which also proposes some ways to accelerate the emulation. However, through chapter 2 we will give a complete example of how a network adapter is virtualized in our reference VMM (QEMU).

In order to further improve the early I/O virtualization tecnhiques, research and products have followed three routes:

- Hardware support in the devices (*virtual functions* and IOMMUs [18]), so that guest machines can access directly and in a protected way subsets of the device and run at native speed.

- Run-time optimizations in the VMM. As an example, [3] shows how short sequences of code involving multiple I/O instructions can be profitably run in interpreted mode to save some VM exits.

- Reduce the expensive operations in device emulation (I/O accesses and interrupts) by designing *virtual* device models more amenable to emulation. This approach is known as *device paravirtualization* and has produced several different virtual device models (vmxnet [16], virtio [13], xenfront [6]), in turn requiring custom device drivers in the guest OS. Synchronization between the guest and the VMM uses a shared memory block, which is used to exchange the state of the communication, while I/O accesses and interrupts are used only for notifications: In this way minimizing the amount of VMExits is easier.

We will analyze device paravirtualization in chapter 4. In section 4.4 we will show that paravirtualization can be introduced with minimal extensions into an existing physical device model.

Chapter 2

Work environment

In section 1.1 we have introduced a classification of Virtual Machine systems. The work presented in this thesis is restricted to same-ISA System Virtual Machines, where the Virtual Machine Monitor is a type 2 VMM. In other words, we will deal with VMs that are able to run an arbitrary OS compiled for the host ISA. The guest OS can in turn provide an execution environment for many user applications. Since the VMM is of type 2, it is implemented as a regular process in the host OS, and can make use of all the OS services. We can therefore access the host physical resources without requiring administrator previleges.
We also restrict our work to VMMs that make use of hardware-based virtualization, because the optimizations we will introduce are particularly effective in limiting the amount of VM switches between the host world and the guest world. Since these VM switches are very expensive with hardware virtualization, the performance gain is going to be significant.

While the assumptions made may appear restrictive, they are not at all. The class of VMMs that we consider is extremely common in the world of computing. These VMMs are used in datacenters and IT departments for server consolidation, application isolation, to provide users/developers with zero-setup computing environments or for other application in which it is not important that the VM computing environment has a different ISA from the host ISA. Moreover, hardware-based virtualization is generally the most efficient CPU virtualization technique, and so it's worth focusing on this case. Several VMM software belonging to the considered class are available. QEMU, VirtualBox, VMWare, Parallels, Windows Hyper-V or Windows VirtualPC are among the most common examples of this kind of VMMs. These software tools are extremely widespread and for this reason performance optimizations in these area are certainly useful.

This said, we have chosen the QEMU-KVM Virtual Machine Monitor for implementations and tests, although our optimizations are relevant to the entire class of VMMs.

A GNU/Linux-based operating system (Archlinux) has been used on the host machine. The guest OS is generally Archlinux, but some tests have been performed with FreeBSD as a guest, too. Although Linux is a kernel and not a complete OS, in the following we will use the expression "Linux OS" to actually mean "GNU/Linux-based OS", for the sake of simplicity.
Since our optimizations concern network performance, we had to choose a network device to work with. The *e1000* class of network devices was chosen, since it is emulated by the vast majority of VMMs and supported by the main OSs (Windows, Linux, FreeBSD).

2.1 QEMU features

QEMU ([1], [5]) is a free, open source and multi-platform type 2 VMM, that makes use of dynamic translation to achieve good emulation performance. QEMU-KVM is a QEMU branch that extends the original software to take advantage of hardware-based virtualization. Whenever possible, QEMU-KVM uses hardware virtualization in order to execute guest code natively. In the following we will use the terms QEMU and QEMU-KVM in an interchangeable manner. At the time of this writing, the QEMU-KVM version number is 1.2.0, and so we will refer to that version.

QEMU is a very flexible tool:

- It supports process virtual machines: by means of dynamic translation it can execute on the host OS a single program compiled for an other ISA. This operating mode is called *User mode emulation*.
- It supports system virtual machines: by means of dynamic translation and hardware-assisted virtualization (when possible) it can emulate full computer systems (including common peripherals), supporting unmodified operating systems. This operating mode (which is the one we are interested in) is called *Full system emulation*.
- It supports various architectures, including IA-32 (x86), x86-64, MIPS R4000, Sun's SPARC sun4m, Sun's SPARC sun4u, ARM development boards (Integrator/CP and Versatile/PB), SH4 SHIX board, PowerPC, ETRAX CRIS and MicroBlaze architectures.
- It can emulate Symmetric Multiprocessing Systems (SMP), making use of all the CPUs that are present on the host system.
- It is able to emulate various peripherals, such as hard disks, CD-ROM drives, network cards, audio interfaces, or USB devices.
- Like similar hypervisors, it is able to provide its VM with network connectivity. The way this can be done will be presented in section 2.2.3.

- It does not normally require administrative rights to run. In our experiments administrative rights won't be necessary.

QEMU is able to emulate the e1000 class of PCI network adapters[1], as well as other network devices (RTL8139C+, i8255x (PRO100), NE2000 (RTL8029), AMD PC-NET II (Am79C7070a)).
Moreover, QEMU supports the *Virtio* framework, that exposes a paravirtualized network device, *virtio-net*, intended to be used for high performance networking. The Virtio paravirtualized soution will be analized in chapter 4.

2.2 QEMU internal architecture

In this section we will illustrate those details of QEMU implementation that is necessary understand in order to implement our optimizations.

2.2.1 QEMU event-loop

QEMU is an event-loop based software, implemented as a single-process multi-threaded application. One thread, referred to as *IOThread*, executes the event-loop, waiting for new events to occur[2]. The waiting routine is a `select()` system call, which is not the most efficient choice on Unix-like systems[3], but is more portable across different platforms.

The file descriptors associated with the `select()` can be associated to regular files, sockets, device files (such as TAP devices), or even special in-kernel objects, such as POSIX timers, signals and eventfds. These file descriptors are used by QEMU to let the VM communicate with the host, and possibly with the rest of the Internet. In other words, they are used for performing the I/O operations requested by the VM. Of course the guest OS still performs I/O operations accessing I/O ports or memory-mapped I/O (MMIO) in its physical address space and is unaware of being emulated.

The QEMU core codebase offers to the QEMU developer an API that can be used to implement devices emulators. In particular, the API provides two useful abstractions: the QEMUTimer and the QEMUBH.
QEMUTimers are one-shot absolute timers, that can be used to execute a callback function at a certain point of time in the future. The callback is always executed by the IOThread, when this recognizes that the deadline has been passed. The QEMUTimers are supported by the Linux OS with a single one-shot relative POSIX

[1] To be more precise, the emulated hardware exposes to the guest OS the PCI device ID of the 82540EM model.

[2] This is implemented in `main-loop.c`

[3] `poll()`, and specially Linux `epoll()` or BSD `kqueue()` are more efficient.

timer[4], which is always (re)armed to expire - waking up the event-loop - at the earliest deadline. The expiration check for QEMUTimers is done at the end of each event-loop iteration, even if the event-loop was waken up for a reason different from the POSIX timer expiration. In the current implementation, moreover, every time the POSIX timer is rearmed, the relative deadline is forced to be greater or equal that 250 μs[5].

QEMUBH is the QEMU abstraction of the *bottom half* concept, widely used in OS drivers implementation. A QEMUBH can be seen as a QEMUTimer that expires as soon as possible. In practice when a QEMUBH is scheduled the event-loop is notified, so that it wakes up (or finishes the current iteration and begins another iteration) and executes all the callbacks of currently scheduled QEMUBHs. Therefore the QEMUBH callbacks are always executed by the IOThread, similarly to the QEMUTimer callbacks[6]. This feature is very important in terms of parallelism, as will be clear in the following chapters.

2.2.2 VCPU Threads

While executing the event-loop, the IOThread handles all the interactions between the VM and the external world. The guest code, however, is executed by one or more threads, that we will call *VCPU threads*. In the current implementation QEMU creates as many VCPU threads as the number of SMP processors specified by the QEMU user. When using hardware-based virtualization (we made this assumption at the beginning of this chapter), a VCPU thread continuously switch between the VM mode and the normal mode (see section 1.2.3).

When the VCPU tries to execute an I/O operation, accessing I/O ports or MMIO, a VMExit happens. A VMExit also occurs when an interrupt is delivered to a VCPU. There could be other type of events that cause a VMExit, but we are not interested in them. On a VMExit the VCPU stops executing guest code and starts executing QEMU code, in order to handle, if necessary, the event that caused the VMExit. After the event has been handled, the VPCU executes a VMEnter and continues to run the guest code from the point where it was interrupted.

Since multiple threads are involved, and all of them - IOThread included - can access the shared structures used by the emulator (e.g. the structures employed to implement the virtual devices), mutual exclusion is required. In the current implementation the mutual exclusion is guaranteed by a single big-lock, called the

[4]Similar mechanisms are used in other OS, or if POSIX timers are not available under the Linux OS.

[5]For more information about the QEMUTimer interface and implementation, refer to the file `qemu-timer.c` in the QEMU project root directory.

[6]For more information about the QEMUBH interface and implementation, refer to the files `qemu-aio.h` and `async.c` in the QEMU project root directory

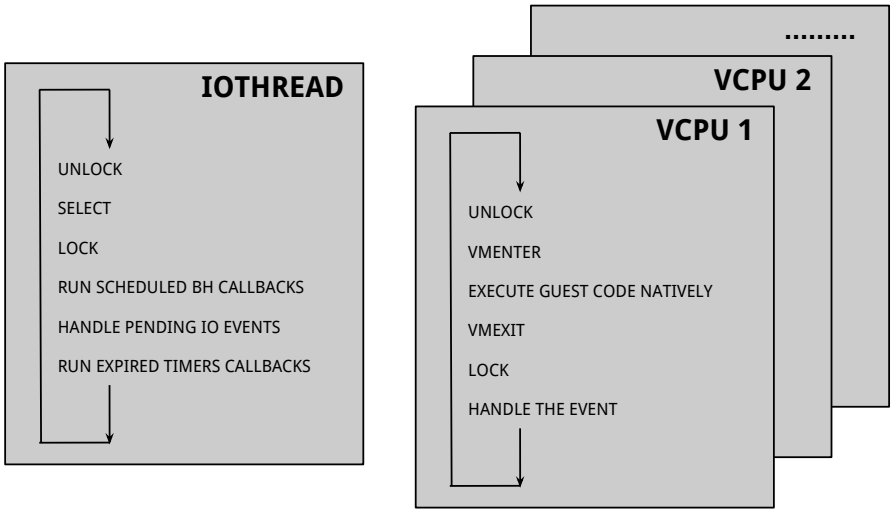

Figure 2.1: Threads of a QEMU process. The IOThread executes the event-loop while each VCPU thread executes a guest CPU.

`iothread lock`.
Therefore, everytime a VCPU VMExits, it has to acquire the `iothread lock` before it can handle the event. After the event has been handled, the VPCU thread release the `iothread lock` and executes a VMEnter instruction. Similarly, the IOthread has to acquire the `iothread lock` every time it wakes up for event handling, and release the lock only when the event-loop iteration terminates.
Putting all together, the figure 2.1 depicts the QEMU thread scheme.

2.2.3 Networking architectures

When running a VM it is of fundamental importance to make possible for the guest to communicate with the outside world using the networking infrastructure, otherwise the VM itself would be an useless computing box.
Since the VM it's a software entity, however, it isn't connected to any real network. Therefore the hypervisor has to provide some form of network infrastructure virtualization, so that the guest OS thinks its (virtual) network device is connected to a physical network and can then exchange packets with the outside.

All the hypervisors cited previously (QEMU included) provide the user with a few virtual network infrastructure modes, so that she can choose the best way to connect her VM. Three modes are commonly employed:

- NAT mode. In this case the guest OS thinks to be physically connected to a completely fake LAN, entirely emulated inside the hypervisor. The VMM usually emulates a DHCP server, a DNS server and a gateway router, so that the guest OS can easily configure its network interfaces and its routing tables to communicate with the outside world. When the guest sends a TCP/UDP packet on the fake LAN, the VMM intercepts the packet, performs address translation (NAT) turning the guest source IP (the guest IP) into the host IP and sends the packet towards its destination using the host OS services (thus the host OS routing tables). The inverse translation is performed when receiving a packet.

  In this way the VM is easily provided with Internet connectivity, but it's not visible from the outside and cannot communicate with other VMs present on the same host. In QEMU this mode is called *Usermode networking*.

- Host-only mode. Also in this case the guest OS thinks to be physically connected to a LAN. The LAN is emulated by means of a software bridge (that emulates a layer-2 network switch), and the VM is connected to a port of that bridge. More VMs can be connected to the same bridge, making inter-VM communication possible. The software bridge can be internally implemented within the hypervisor, or can be an external software bridge.

  Whit QEMU this mode can be set up on a Linux host using the in-kernel bridging and TAP interfaces. Each VM is assigned a TAP interface where it can write/read ethernet frames, and all the TAPs are bridged together to the in-kernel bridge. In this way a frame sent by the guest is written by a QEMU instance to its associated TAP and is therefore routed by the bridge to the correct destination TAP. The receiving QEMU process can then read the frame from the TAP and push it to its VM. In this case no DHCP or DNS server is emulated, and you have to configure yourself the network of each VM[7]. Since the software bridge itself has its separate network interface, also the host can communicate on the LAN.

- Bridged mode. This mode is an extension of the host-only mode. The only difference is that a physical host network interface, connected to a real network, is also bridged to the VMs LAN. Since the physical interface becomes a bridge port, the host can still access the physical network through the software bridge interface. In this way the host can share its connectivity with all the VMs connected to the software bridge. If the physical interface is connected to a LAN, the VMs LAN appears to be part of the physical LAN.

Clearly the NAT mode is not interesting with respect to our goals, since it is only intended to be a way the VM can easily obtain Internet connectivity, and it's not intended to be a flexible or efficient networking mode. Instead we will consider

[7]The configuration can be static or you can run a DHCP server on one of the VMs connected to the bridge.

host-only mode, since we are intersted in optimizing the communication performance between two VMs on the same software bridge or between a VM and the host bridge interface. In this work it would make no sense considering to bridge also the host physical interfaces (bridged mode), because optimizing the performance of a real network adapter it's not among our goals.

2.2.4 Network frontend and backend

In order to implement a specific networking architecture, the QEMU implementation includes a degree o separation, namely an interface, between the piece of code that emulates the network adapter and the code that provides access to the chosen networking model. This is done because the two subsystems are completely independent, and you can easily combine every virtual network adapter with every networking access mode.
Using the QEMU terminology, the network device emulation is also called *network frontend*, and the networking access mode is called *network backend*.

In our case the network frontend is e1000. The e1000 frontend is implemented within the file hw/e1000.c (in the QEMU project root directory). This source file is the only one that contains code which is specific to the e1000 class of networking devices, exporting to the rest of the system the same interface exported by the other network devices.

The network backend can be

- "user", which implements Usermode networking.
- "tap", which is an implementation of host-only/bridged networking that relies on TAP devices and in-kernel bridges.
- other implementations of host-only/bridged networking that use a different software bridge solutions, maybe in conjunction with TAPs and in-kernel bridges. The "VALE" backend ([11]) is an example of high performance alternative implementation of the host-only/bridged networking access mode.

Frontend and backend can be seen as two peers connected to each other that are able to exchange ethernet frames. Each peer exports a `receive` method that accepts an ethernet frame as argument[8]. That method will be invoked by the QEMU network core when the other peer wants to send a frame. Moreover, each peer can optionally export a `can_receive` method that is called right before the `receive` method, to make sure that the peer is willing to receive a frame (e.g. it has enough space). If `can_receive` returns 0, the corresponding `receive` method is not called and the packet sent is appended to an internal queue. If `can_receive` returns 1, the `receive` method is invoked. A peer can receive all the packets queued into its

[8]The ethernet frame can be specified with an address and a length, or with a scatter-gather array.

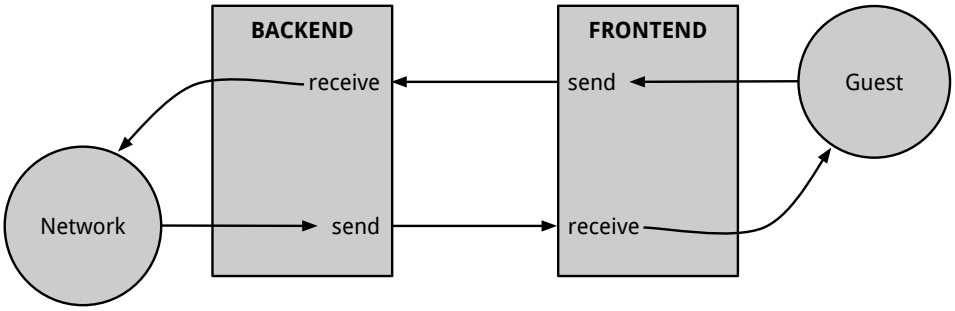

Figure 2.2: Interface between a QEMU network frontend and a QEMU network backend. Each peer can send network packets to the other peer calling the `qemu_send_packet()` function.

internal queue by calling `qemu_flush_queued_packets()`. If a `can_receive` method is not defined, however, the `receive` method is always called.

In this way, when a frontend wants to send a frame to the network, it invokes the `qemu_send_packet` function provided by the QEMU networking API, that will invoke the `receive` method exported by the backend. The backend `receive` method will push the frame onto the network, in a way that is specific to the backend itself. On the other direction, when the backend gets a frame from the network, it invokes the `qemu_send_packet` function, that will in turn invoke the `receive` method exported by the frontend. This method will push the frame into the guest system in a way that is specific to the network device model.
The frontend-backend interface is depicted in figure 2.2.

2.3 The e1000 class of network adapters

In this section we will illustrate some features and some details about the inner working of an e1000 ethernet network adapter. Once again, we will only describe those aspects that are relevant to our goals. The complete specification can be found in [7].

Since the network communication is extremely important in the IT world, the market constantly pushes hardware vendors to produce high performance, low-power, fully featured flexible, network adapters.
As a result, modern network devices are very complex. To get an idea of this complexity, one can observe that a device can implement more then a hundred of software-visible registers. Complexity is the price to pay in order to get high flexibity and several useful features. Flexibility helps the IT administrator to tune the device parameters in order to find the right tradeoff between performance and power consumption, throughput and latency, and the like. The rich set of features

the device come with helps to adapt the device to different usage patterns and allows for performance optimizations (expecially for TCP/IP traffic) when offloading capabilities are present. Hardware offloading features are supported by virtually every recent network adapter.
The most common feature are

- Checksumming offload. When this feature is present, the device computes in hardware the checksums required by the main Internet protocols, such as IP, TCP and UDP. This saves the OS to do this work in software, which could be very expensive, expecially with checksums that are computed also on the payload and not only on the protocol header (e.g. TCP and UDP checksums).
- TCP Segmentation Offload. When this feature is present, the device is able to do the TCP segmentation in hardware, splitting a TCP segment over multiple ethernet frames. The segmentation is necessary because the real MTU of a TCP connection is almost never greater than 1500 bytes (the ethernet original MTU), but the TCP window is commonly greater than that value. The OS is then forced to do the splitting. With the feature present, however, the OS can send to the device driver a frame containing a TCP segment which is greater than the MTU[9]. The device driver can pass that frame to the adapter that performs the segmentation in hardware and sends multiple ethernet frames. Apart from the obvious speed-up obtained because the operation is done in hardware rather then in software, this mechanisms has an important positive side effect. The network overhead necessary to traverse the TCP/IP stack is suffered only once, for a big TCP packet, instead of once for each MTU-sized fragment. This clearly amortize the kernel per-packet overhead.
- Scatter-gather capability. When this feature is present, the device is able to send a frame that is stored in multiple non contiguous fragments in the machine physical memory. Therefore the OS is not forced to linearize (gather) the fragments, avoiding the copy overhead. This is useful specially when the OS wants to send large packets. In other words the device is gather-capable.

The e1000 adapters have this three offloading capabilities.

2.3.1 Data interface with the driver

Being high performance devices, modern network adapters rely on Direct Memory Access (DMA) operations and interrupts.
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[10]. Once the device knows where the frame is, it can directly access the physical memory[11] and send it on the wire. More commonly, the frame is DMA'ed into an

[9] Up to 64KB with the Linux kernel.
[10] When dealing with fragmented frames, the driver has to specify somehow a scatter-gather array.
[11] In this example we assume there isn't any IOMMU.

internal buffer for further processing before being sent on the wire, but this is just a detail.
When the adapter receives a frame from the wire, it has to store it in the machine physical memory. For this reason, the device driver has to tell the adapter where it can store incoming frames, before the frames actually comes. If the adapter doesn't know where to put incoming frames, it cannot accept them.

It is clear that there must be a well-defined interface between the driver and the device. This interface is known as *ring*. A ring is a circular array of *descriptors* that are used to exchange address/length information[12]. A network adapter has at least two rings. The first one is the *TX ring* and it is used with outgoing frames, whereas the second one, the *RX ring*, is used with incoming frames. Network adapter can have multiple TX/RX rings with possibly different policies and priorities, so that one can do some traffic engineering.
However, the e1000 adapter model emulated by QEMU has one TX ring and one RX ring. The array length, namely the number of descriptors, can be chosen by the driver. In e1000 it must be a power of two and not more than 4096.

TX ring

The e1000 TX ring is an array of *N* TX descriptors. Each TX descriptor is 16 bytes long and contains the address (in physical memory) and the length of a frame (or a frame fragment) that is going to be sent or that has already been sent. The descriptor contains also flags that can be used to specify options, and status bits.
Two index registers are implemented for the synchronization between the driver and the adapter: The TDT register (Transmit Descriptor Tail) and the TDH register (Transmit Descriptor Head). These are *index* registers since their value are array indexes with respect to the TX ring.

At the start-up, TDT and TDH are initialized by the driver to the same value, usually 0. When the driver wants to send a new frame, it writes the physical address and the length of the frame in the TX descriptor pointed by the TDT register and then increments the TDT register itself. Since the descriptor array is circular, the TDT must be set adequately.
When the adapter recognizes that TDH is different by TDT, is knows that there are new frames to transmit, and start processing the descriptor pointed by TDH. A write access to the TDT register is therefore the way the device driver notifies the adapter that there are new frames ready to be sent.
For each descriptor to process:

1. The frame pointed by the decriptor is sent on the wire.
2. The TX descriptor is written back in order to set the DD (Descriptor Done) bit that indicates the TX descriptor has been processed.

[12] Being an array a ring is a contiguous zone in the physical memory.

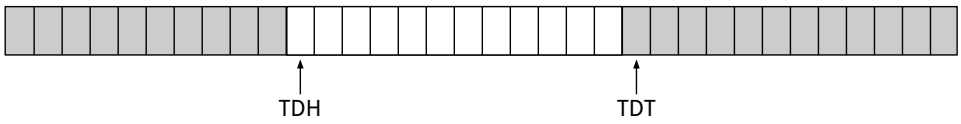

Figure 2.3: The e1000 TX ring with its index registers. The grey area contains software-owned descriptors, while the white area contains hardware-owned descriptors. Index registers are used for synchronization between software and hardware.

3. The TDH register is incremented circularly.

The adapter stops the processing only when TDH == TDT, e.g. there are no more descriptors to process.
When the driver increments the TDT, the descriptor previously pointed (the one that has just been written) is committed to the hardware, and the hardware owns it until the decriptor is processed.
Therefore, in each moment the ring is partitioned in two contiguous parts: The descriptors owned by the hardware, which are waiting to be sent on the wire, and the descriptors owned by software, which are free to be used by the driver to commit new frames.
In order to prevent the index registers to wrap around, the driver should never use a TX descriptor if this is the very last free TX descriptor. This happens when TDT is shuch that incrementing it circularly would cause TDT == TDH. When the TX ring is in this state (TX ring full), the driver should stop transmitting. The transmission can be enabled again when the hardware process some descriptors, incrementing TDH.
The figure 2.3 depicts the TX ring with its index registers.

Multi-descriptor frames Sometimes a frame is not stored contiguously in physical memory, or it's bigger than 4KB. In these cases the frame to transmit cannot be specified with a single TX descriptor, but many consecutive TX descriptor must be used. The last TX descriptor describing a frame must have the EOP (End Of Packet) bit set.

Context descriptors There are actually two generations of TX descriptors: Legacy descriptors and Extended descriptors, the latter being the most recent ones. When extended descriptors are used (this is normally the case), hardware offloading requests can be specified for each TX frame to send by putting a so called *context* descriptor in the TX ring before inserting the regular data TX descriptor(s). Linux device driver makes use of extended TX descriptors and context descriptors.

RX ring

The e1000 RX ring is an array of *N* RX descriptors. Each RX descriptor is 16 bytes long and contains the address (in physical memory) and the length of a frame that has been received by the adapter, or only the address of a memory location that can be used by the adapter to store an incoming frame. The descriptor contains also flags that can be used to specify options and status bits. Two index registers are implemented for the synchronization between the driver and the adapter: The RDT register (Receive Descriptor Tail) and the RDH register (Receive Descriptor Head). These are *index* registers since their value are array indexes with respect to the RX ring array.

At the start-up, the driver initializes RDH and RDT to 0. At this point, the adapter still doesn't know of any memory buffer where it can store incoming frames, so it cannot receive anything. To give the hardware memory buffers to work with, the driver puts the physical address of a memory buffer[13] in the RX descriptor pointed by RDT and increments RDT circularly. Writing to the length field of the RX descriptor is useless, since this value is not used by the device. It's important to note that the size of the memory buffer must be greater or equal then the maximum frame length, since we cannot know in advance how long a future incoming frame is going to be.
A write access to the RDT register is therefore the way the device driver notifies the adapter that there are new memory buffers ready to be used to store incoming frames.
If the adapter sees that RDH is different by RDT, it recognizes to have memory buffers available, and starts accepting incoming frames. When a new frame arrives from the wire, the adapter

1. Fetches the RX descriptor pointed by the RDH register.
2. Copies the frame to the address contained in the RX descriptor.
3. Writes back the descriptor in order to write the length of the received frame, and to set the DD (Descriptor Done) bit. The DD bit indicates that the RX descriptor has been used to receive a frame.
4. Increments the RDH register circularly.
5. If programmed to do so, the device would normally sent an interrupt (see section 2.3.2), in order to tell the driver there are new received frames ready to be pushed to the kernel network stack.

When the driver increments the RDT, the descriptor previously pointed (the one that has just been written) is committed to the hardware, and the hardware owns it until the decriptor is used.

[13] How the memory buffer is allocated depends on the OS and on how the driver is implemented.

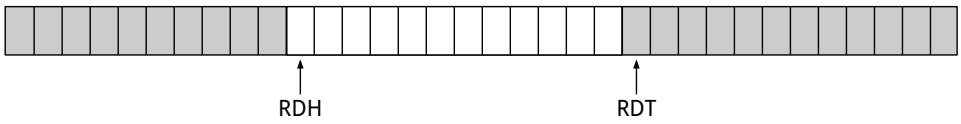

Figure 2.4: The e1000 RX ring with its index registers. The grey area contains software-owned descriptors, while the white area contains hardware-owned descriptors. Index registers are used for synchronization between software and hardware.

Therefore, in each moment the ring is partitioned in two contiguous parts: The descriptors owned by the hardware, which can be used to store new incoming frames, and the descriptors owned by software, which are unused or point to received frames ready to be pushed to the network stack.
Similarly to what happens the TX ring, in order to prevent the register indexes to wrap around, the driver should never increment the RDT register if the increment would cause RDT == RDH. When this situation happens (full RX ring) the driver should stop giving memory buffers to the adapter. When new frame are received, the hardware increments RDH, and so it is possible to increment RDT again.

The interrupt routine should then push the arrived frames to the kernel and provide the adapter with more memory buffers (incrementing the RDT), otherwise the adapter cannot accept more incoming frames. A common strategy is to try to keep the RX ring always full. In order to do this, at the startup the driver writes $N - 1$ RX descriptor with the address of $N - 1$ memory buffers, and set RDT to $N - 1$, so that the ring is full. Every time an interrupt arrives, the interrupt routine pushes the received frames to the kernel and replenish the ring, making it full again. This strategy avoid situations in which the adapter is forced to reject incoming frames because it has no memory buffers available.
The figure 2.4 depicts the RX ring with its index registers.

2.3.2 Interrupts generation

The e1000 network adapter can generate interrupts for different reasons, but we are interested in two types of interrupts:

- TX interrupts: these interrupts are generated when the transmission of one ore more hardware-owned frames completes. Each TX descriptor has a bit flag (Report Status, RS) that can be set to specify if the hardware has to send an interrupt as soon as the transmission of the associated frame is complete. In every case an interrupt is always sent when the TX ring becames empty (TDH == TDT). The interrupt routine, depending on the OS and the driver implementation, may execute cleanup operations on the descriptors that have

been processed and mark them as free so that they can be used to commit new frame to the adapter.

- RX interrupts: these interrupts are generated after the hardware has received (and stored in physical memory) an incoming frame, in order to notify the driver that it can send the frame to the kernel network stack. When the frame is sent to the kernel, it will find the right way to its destination, that can be anything (trash included). Assuming the destination is a user process, what the OS does is just appending the packet to a queue associated to the receiving socket. At this point the sleeping user process is notified and can be scheduled to complete the read operation from the socket queue.

There is a big concern when dealing with RX interrupts. If the device doesn't limit them, they act as a source of uncontrolled load for the CPU. The receiving machine, in fact, is forced to serve incoming RX interrupts even if the RX interrupt rate is very high. Since we are dealing with high performance devices, we would like to be able to receive up to 1 Mpps (or even more). The overhead involved in interrupt handling is generally quite high, and so each interrupts has 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 process actually receive it.
If each packet receive generated an interrupt, we would have to handle up to 1000000 interrupts per seconds, which something that would completely stall our machine. When the interrupt rate is too high, in fact, the machine spends almost all the time serving interrupts[14], and there is no time left for other things to happen, e.g. for user processes to read the received packets. This is a quite bad situation, because the CPU is 100% utilized, but the machine, on the whole, cannot do any useful work (this is the *livelock* problem).
The situation could be slightly better if the machine has more than one CPU, but still it's not good for the CPU servicing the interrupts to spend too much time in interrupt handling overhead.

Cleary, this problem can only be solved if the device somehow collapses RX interrupts, raising an interrupt every batch of received frames, let's say 100 frames per batch, and not every single frame. In this way the interrupt rate is 100 times lower, and the interrupt overhead cost is amortized over 100 frames. In every case the device must guarantee that a RX interrupt is eventually sent after a period of inactivity, even if it is still waiting for a 100-frames batch to be completed, because the device cannot know when the next frames are going to come.
These and similar mechanisms are known as *interrupt mitigation* or *interrupt moderation*, since they tries to moderate the interrupt rate.
Interrupt moderations is commonly applied also to TX interrupts (or to all the interrupts in general). The TX interrupt rate can be controlled because the driver can

[14]The interrupt handling is by its nature an high priority task with respect to normal process execution.

control (and limit) the TX rate. Neverthless it is convenient for the driver to take advantage of the interrupt-rate-limiting hardware capabilities rather than implement a similar feature in software[15].

The e1000 class of network adapters implements two interrupt moderation mechanisms.

The older moderation mechanism

The older mechanism is supported by the registers TIDV (Transmit Interrupt Delay Value), TADV (Transmit Absolute Interrupt Delay Value), RDTR (Receive Delay Timer Register) and RADV (Receive Interrupt Absolute Delay Value), where each register is provided with a countdown timer.

TIDV and TADV, when properly set, can be used to provide TX interrupt moderation on a per-descriptor basis.
When the adapter processes a a TX descriptor, if the RS bit and the IDE bit[16] are set, an interrupt is not raised immediately, but the TIDV timer is armed (or rearmed) with the value specified in the TIDV register itself. When the TIDV timer expires, an interrupt is raised in order to inform the driver about the TX completion of one or many frames. The TIDV register can therefore be used to coalesce TX interrupts. However, it might be necessary to ensure that no completed transmission remains unnoticed for too long an interval in order to ensure timely release of TX buffers. The TADV register has been added for this purpose. Like the TIDV timer, tha TADV timer only applies to TX descriptor where both the RS bit and the IDE bit are set. This register can be used to ensure that a TX interrupt occurs before a predifined time interval after a transmission (whatever) is completed. The time interval can be specified in the TADV register itself. After each TX descriptor is processed, the TADV timer is is armed with the value specified in the TADV register only if it is not already running. When the timer expires, a TX interrupt is generated.

RDTR and RADV, when properly set, can be used to provide RX interrupt moderation on a per-received-frame basis. The RDTR timer is armed (or rearmed) immediatly after a new packet is received and transferred to physical memory, using the interval value specified in the RDTR register. When the RDTR timer expires, an RX interrupt is raised, and the timer is cleared. The RDTR register can therefore be used to coalesce RX interrupts, in the very same way the register TIDV is used to coalesce TX interrupts.
Also in the RX case it may be necessary to ensure that no receive remains unnoticed for too a long interval. The RADV register deal with this problem, in the very same way the TADV register does, so we won't explain the mechanism again.

[15] With e1000, an TX interrupt moderation mechanism could be implemented using the RS bit.
[16] The Interrupt Delay Enable bit (IDE), is another bit flag in the TX descriptor, like the RS bit.

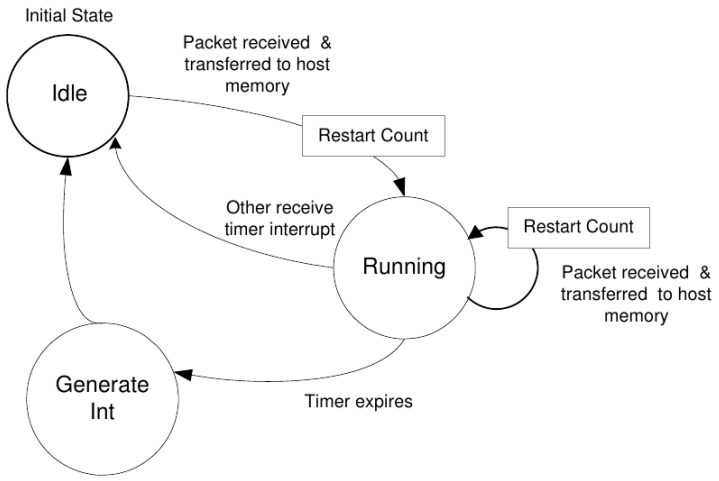

Figure 2.5: State diagram of the e1000 RDTR timer (Packet Delay Timer).

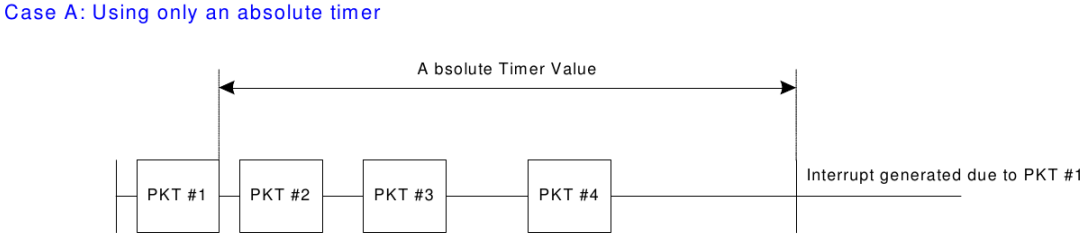

Figure 2.6: RX moderation example that makes use of the e1000 RADV timer.

Figure 2.5 shows a state diagram associated to the RDTR timer, while figure 2.6 depicts an example about the RADV moderation mechanism.

The newer moderation mechanism

Although the older mechanisms allows for fine grained control over the moderation, especially on the TX side, most of the times the required moderation functionality is way simpler. For this reason a more direct moderation mechanism has been added, which is implemented through the ITR register (Interrupt Throttling Register).
If the driver sets this register to a value δ, the board ensures that δ is the minimum inter-interrupt interval, regardless of network traffic condition, and the interrupt type. In other words, every time an event happens that requires an interrupt (e.g. TX completion or RX completion) the board raise an interrupt as soon as possible, while meeting the ITR inter-interrupt delay constraint.

The ITR mechanism, when used, applies to all the interrupts. The Intel manual

strongly recommend avoiding the RDTR and RADV register, and use the ITR instead.

2.4 QEMU e1000 emulation

The e1000 frontend is implemented in QEMU through a single source file[17] (see section 2.2.4).
A small part of this code contains declarations and routines necessary to register and initialize/uninitialize a new type of PCI Ethernet device within the rest of the emulator. In this way one or more instances of the e1000 network device can be included in a VM when launching QEMU[18].
When registering a new PCI device, it is necessary to describe the I/O or MMIO regions that the device implements, registering new PCI BAR registers. Each BAR register correspond to a different I/O or MMIO region. When registering a new BAR, you can specify two callback functions that are invoked whenever the guest code access (reads or writes) a location in a region.
The e1000 emulation code registers a MMIO region and an I/O region. The I/O region isn't actually used, whereas the MMIO region maps all the registers the e1000 device implement, such as TDT, TDH, RDH and so on.
A couple of statically defined dispatch tables, one for the read accesses and the other for the write accesses, are used to associate a different function to each register. In this way one can (potentially) associate a different read callback and a write callback to each e1000 register. The emulation of a device is basically achieved with this per-register functions. Of course register can share the same functions or have no callbacks at all.

Let's see in more depth how register callbacks are actually invoked.
When a VCPU is executing the guest code, it may try to access a MMIO location corresponding to the e1000 PCI device, namely an e1000 register. This usually happens when executing the e1000 device driver. The accessing instruction causes a VMExit to occur, and the VCPU thread switches from the guest world to the host world. After the `iothread lock` has been acquired, the QEMU code analyzes the VMExit reason and understands that the VMExit was caused by a MMIO access. It then uses the physical memory address involved in the MMIO access to dispatch the event to the right PCI memory region. In our case, the read (write) callback registered with the e1000 MMIO BAR is invoked. This callback uses the address to access the e1000 read (write) dispatch table and invokes the register-specific callback.
The register-specific callback emulates the side effects associated with the register access. For instance a write to the TDT register will cause the emulated hardware to process all the pending TX frames (see section 2.3.1).

[17] hw/e1000.c in the QEMU project root directory
[18] This is done through the `device` option. E.g. `qemu-kvm -device e1000 ...`

After the callback returns, the `iothread lock` is released and a VMEnter is executed so that the VCPU switches back to the guest world.

This said, we won't describe in detail all the aspects involved in e1000 emulation. Instead, we will outline the frame transmission and frame reception process, focusing on notifications, scheduling and memory copies.

2.4.1 TX emulation

As stated in section 2.3.1, a write to the TDT register is the way the driver notifies the hardware that new TX frames are ready to be processed. The TDT register write callback updates the TDT value and then calls the `start_xmit` function. This function is a while loop that processes all the committed TX descriptors, starting from the descriptor pointed by the TDH register. After each descriptor is processed, the TDH register is incremented circularly. The while loop exits when TDT == TDH, e.g. when there are no pending TX descriptors.

TX descriptors processing includes the following actions:

1. The TX frame corresponding to the TX descriptor is copied from the guest memory to a local buffer.
2. The TCP/UDP checksum is computed. Recall that e1000 devices are able to compute checksums in hardware, and so the OS driver expects the emulated hardware to be able to do it.
3. The `qemu_send_packet` function is invoked in order to pass the frame to the network backend. With the TAP backend, a `write` system call is invoked to pass the frame to the TAP device associated with the backend.
4. The descriptor is written back to the guest memory in order to report that the TX descriptor has been processed.

When all the pending TX descriptors have been processed, the interrupt pin associated with the e1000 adapter is raised (if the previous pin state was low), sending an interrupt to the guest. The interrupt moderation registers (see section 2.3.2) are not implemented.

It's very important to observe that all these operations (frontend and backend) are performed by the same VCPU thread that tries to execute the TDT write operation in the guest, causing a VMExit. This means that when the guest has only a VCPU (no SMP) the emulation of a transmission cannot be done in parallel with the guest, being done synchronously with the VCPU thread.

2.4.2 RX emulation

When a receiving guest is waiting for new frames to come, the IOThread is blocked in the `select` system call, waiting for the TAP file descriptor to be ready. When a frame arrives to the TAP backend, the `select` returns and invokes the TAP network backend. The backend executes a `read` system call on the TAP device, extracting the incoming frame, and invokes the `qemu_send_packet` function that passes the frame to the e1000 frontend, invoking the `receive` method (`e1000_receive`).

According to what has been presented in section 2.3.1, the `e1000_receive` method performs the following actions:

1. If RDH == RDT, there are no RX memory buffer available, and the incoming frame must be dropped. Otherwise go to step 2.
2. The RX descriptor pointed by the RDH register is fetched from the guest memory.
3. The incoming frame is copied to the guest memory location at the address specified in the RX descriptor.
4. The RX descriptor is written back to guest memory, in order to report the length of the received frame and set the DD bit.
5. If not already high, the e1000 interrupt pin is raised.

Differently from the TX emulation, here all these operations (frontend and backend) are executed by the IOThread, that can run in parallel with the VCPU thread (or the VCPU threads) executing the guest code.

When the `qemu_send_packet` function returns, the backends tries to read another frame from the TAP[19]. If there is another frame to process, the `qemu_send_packet` is called also on this frame. This process stops when no frames are ready to be read from the TAP, or when the packet is dropped by the frontend.

2.5 Linux e1000 device driver

In this section we will describe some details about the e1000 device driver in the Linux kernel, which is implemented as a kernel module. We will only illustrate those aspects that are relevant to our goals. The driver source code can be found in the directory `drivers/net/ethernet/intel/e1000` in the Linux kernel project root directory.

[19]The `read` system call is non-blocking, since the IOThread is executing an event-loop, and so only the central `select` system call is allowed to block.

2.5.1 Interface with the network stack

The Linux kernel API has a specific network API that the network device drivers use to exchange network packets with the rest of the kernel.
With these API, the network driver can register a new network interface within the kernel, associating a name to it. The list of all the network interfaces currently registered within the kernel can be seen using the `ifconfig` utility (e.g. `$ ifconfig -a`) or similar tools. The registering function, `register_netdev`, requires as input argument a `netdev` structure that abstracts the registering network adapter. This structure contains all the information necessary to the kernel to communicate with the adapter.

The most important `netdev` fields are:

- `name`, which is a string that univocally identifies the new network interface in the system.
- `netdev_ops`, which is a structure containing the methods that the new network interface exports to the kernel (see below).
- `watchdog_timeo`, which specifies the minimum time interval that the kernel should wait after a transmission is commited to the device driver before deciding that something could be wrong. When a transmission is committed, the watchdog timer associated with the network interface is started. When the driver knows that the hardware it done with the committed frame, it release the associated TX buffer (in e1000 this can be done in the routine that handles the TX interrupt). On the release, the watchdog timer is cleared. If the timer fires, the `ndo_tx_timeout` method is called, so that the driver can handle potential hardware hangs.
- `hw_features`, which is a bitmask that specifies to the kernel the features offered by the hardware that can be activated or deactivated by the user. The e1000 device driver specifies, among the others, the `NETIF_F_HW_CSUM` (checksumming capability) feature, the `NETIF_F_SG` (scatter-gather capability) feature and `NETIF_F_TSO` (TCP Segmentation Offload capability).
- `features`, which is a bitmask that represent the currently active features. This mask can be initialized to the same value as the `hw_features` field, but can be modified by the users to set/unset features and is fixed by the kernel in order to meet feature constraints[20].
- `dev_addr` and `perm_addr`, which contain the hardware address of the network adapter. As far as we are concerned, these two fields are set to the same value. In the e1000 adapter, the hardware address is read from an on-board EEPROM.

[20]The features are not independent on each other within the kernel.

The netdev_ops contains several methods, but we are intersted only in a few ones:

- ndo_open, which is called when the system administrator decides to bring the network interface up. This can be done using the ifconfig utility (e.g. #ifconfig eth0 up) or similar. The driver should react to this request by setting up all the resource (software or hardware) necessary for the adapter operation. The e1000 ndo_open method allocates and initializes all the resource necessary for TX/RX operation (e.g. RX and TX rings), initializes the registers, enables the e1000 interrupts and invokes the netif_start_queue function, which tells the kernel it can start invoking the ndo_start_xmit method (see below).

- ndo_close, which is called when the system administrator decides to bring the network interface down, This can be done using the ifconfig utility (e.g. # ifconfig eth0 down or similar). When receiving this request, the driver can release all the resource allocated by the ndo_open method.

- ndo_start_xmit, which is invoked by the kernel when it wants to transmit a new frame. The first argument is a pointer to a sk_buff structure, which contains all the information related to the frame to transmit, including the frame itself. The sk_buff structure is central to the Linux kernel networking, since all the layers in the network stack perform actions on this structure.

On the receive side, the kernel networking API provides the network driver with a function (netif_rx()) for passing a received frame up to the network stack, where the frame can find its way to its destination. The frame is handed off to the network stack in a sk_buff structure. When receiving a frame from the adapter, therefore, the driver has to build a sk_buff structure around the frame received and call the function netif_rx() with the structure built as parameter.

The NAPI interface

The receive interface described earlier is known as the "old" interface. The netif_rx is intended to be invoked directly by the interrupt routine associated with an RX interrupt.
However, as we pointed out earlier (see 2.3.2), the RX interrupt are source on uncontrolled load and must be moderated when the interrupt rate is too high. This can be done by hardware mechanisms, but a complementary software approach can still be useful. Moreover, a software interrupt moderation can be fundamental if hardware moderation is not provided by the network adapter.

The software approach is based on the concept of *polling*. Interrupt operation was introduced in the early days of computing when the computer architects realized that when dealing with devices through polling most of the CPU cycles were

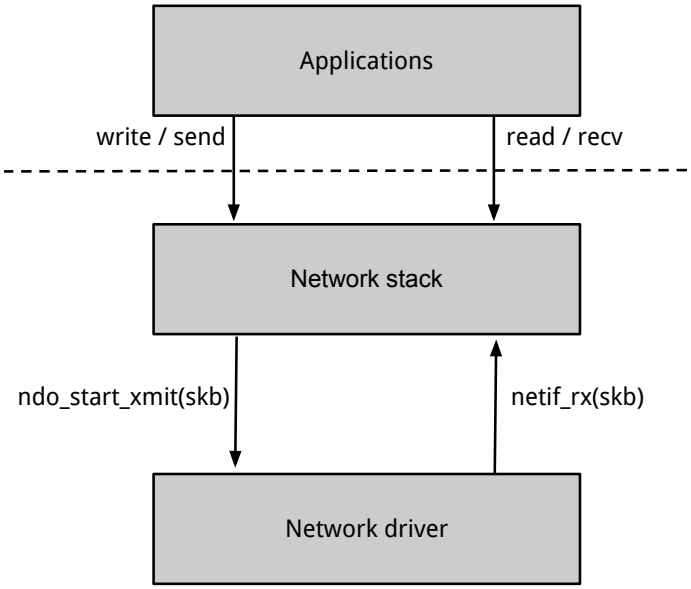

Figure 2.7: On the bottom the interface between the network device driver and the Linux network stack. On the top the interface between user applications and the network stack.

thrown away because of busy waiting. Interrupt operation makes it possible to avoid busy waiting, even if it carries with it a fixed overhead, due to the context switches and scheduling/dispatch operations, that is fairly high.
Nevertheless, if there are (almost) always hardware I/O events to handle, busy waiting is not a problem, since there is (almost) nothing to wait for. Polling can therefore be used in those situations where the input load is very high, since each time we want to check if there are more hardware events to we are likely to find them. The big advantage of polling operation over interrupt operation is that the former does not have any fixed overhead.

In conclusion, one cannot say in general that interrupt operation is better than polled operation or the other way around. When the hardware event rate (e.g. frame reception rate) is low enough, it's worth paying the fixed cost associated with interrupts in order to be sure that there is an event to handle. When the event rate is high enough, however, busy waiting it's cheaper than interrupts, because the average number of cpu cycles wasted to busy wait for the next frame is way lower than the cpu cycles wasted for the interrupt overhead.

The NAPI (New API) interface has been designed with these considerations in

mind. When a RX interrupt arrives, the driver interrupt routine can decide to switch to polling mode, and not to handle the received frame directly. This is done simply disabling the interrupt on the network adapter and scheduling a NAPI context. The latter action is done invoking the function `napi_schedule`. The NAPI context is going to be executed in a kernel thread context different from the interrupt context: This is basically another form of deferred work.
When the NAPI context is scheduled it executes the NAPI polling function registered by the driver through the function `netif_napi_add` (this function registration could be done in the `ndo_open` method).
The NAPI polling function is in charge of doing the receive work[21]. Of course the polling function is intended to process more RX frames. In order to prevent the polling function to monopolize the CPU, however, there must be a limit on the work done by the polling function. This is the reason why the polling function is invoked with a `budget` input argument, that specifies the maximum number of RX frames to process. If the budget is comsumed entirely, the polling function should return, so that the NAPI context will be rescheduled in the future: However, the interrupts are not enabled. In these way the NAPI keep polling the device as long as there is RX work to do.
If the budget is comsumed only partially, however, the polling function should assume that it's not worth polling again, and switches back to interrupt operation. This is done calling the `napi_complete` function, and then reenabling the adapter interrupts. The NAPI context won't be scheduled again until the next interrupt routine calls the `napi_schedule` function the next time.

In the end, with the NAPI interface the driver is able to switch between interrupt operation an polled operation depending on the current incoming traffic conditions, providing a form of interrupt moderation that can greatly increase the receive throughput.
The e1000 driver currently use the NAPI interface instead the older one.

2.5.2 Interface with the PCI subsystem

Since an e1000 network adapter is a PCI device, the first thing the driver has to do when the e1000 kernel module is loaded is registering a new PCI driver within the kernel PCI subsystem. The registration is done through the function `pci_register_driver`, which accepts a `pci_driver` structure as input argument. The `pci_driver` structure contains all the information useful to the Linux PCI subsystem to manage the PCI device.

The most important fields of the `pci_driver` structure are:

- `id_table`, which is a list of all the PCI devices managed by the driver. A PCI device is identified by a vendor ID and a device ID.

[21] or the interrupt work in general

- `probe`, which is a method that the PCI subsystem invokes when it detects (e.g. by PCI enumeration) that a new PCI device is attached to the PCI bus. The PCI subsystem invokes the probe method of the driver that manages the specific PCI device detected. The e1000 probe method initializes, configures and reset the board, an then registers a new network interface (see section 2.5.1).

- `remove`, which is a method that the PCI subsystem invokes to alert the driver that it should release the PCI device, because of a Hot-Plug event or because the driver is going to be removed from memory. The e1000 remove method undo all the operations done by the e1000 probe method, disabling the adapter operation.

Once the PCI subsystem invokes the e1000 probe method, a new e1000 network interface is registered within the kernel and is ready to be used.

2.5.3 TX operation

As outlined in section 2.5.1, when the network stack decides to send a packet through the e1000 adapter, the e1000 `ndo_start_xmit` method is invoked. What the driver has to do is extracting the frame data and other useful information from the input `sk_buff` structure, and commit the frame to the adapter. The first byte of the frame is stored at the address contained in the `data` field of the structure.
The input `sk_buff` can be *linear* or *non-linear*. If linear, the frame is a stored in a contiguous memory area[22]. If non-linear, the frame is not contiguous, but is stored as a collection of contiguous fragments (e.g. is specified by a scatter-gather array).

In order to write the TDT register only when necessary, the driver has a shadow variable, `tx_next_to_use`, that is intended to be used in place of the TDT when possible. The TDT register is updated with the variable content only when the driver wants to commit a frame to the hardware. In this way is never necessary to read the TDT register, since one can read the shadow variable instead.
Similarly, there is a shadow variable, `tx_next_to_clean`, also for the TDH register. The shadow variable is incremented once for each used TX descriptor that has the DD bit set (see below). In this way it's never never necessary to read the TDH register, since one can read the shadow variable instead[23].
Even though the driver wasn't written with virtualization problems in mind, these shadow variables are extremely useful, because accessing a register cause an expensive VMExit, while accessing memory is done at native speed.

Moreover, in order to keep trace of per-descriptor information that can be used to release the resources when the transmission is done, the driver mantains an array

[22] Being kernel *logic* addresses, the frame is contiguous both in virtual and physical memory.

[23] Recall that the driver normally does not need to write the TDH register, except for register initialization.

parallel to the TX ring, the tx_buffer_info array. As an example, some elements in this parallel array contain a valid pointer to a sk_buff structure passed by the kernel. Since this structure (data included) is dynamically allocated, the driver has to free its memory, but only when it is sure that the frame contained has actually been sent on the wire. As we will see, this cleanup is done by the TX interrupt routine.

In more detail, the e1000 ndo_start_xmit method does the following:

- Checks if there are enough free TX descriptors for the frame to send. If not, returns immediately reporting the adapter as busy. Since the sk_buff can be non-linear, we have to use a different TX descriptor for each fragment. Moreover, an additional context descriptor may be necessary in order to make a offload requests on the current frame (see 2.3.1).
- If necessary, inserts a new context descriptor in the TX ring location pointed by the tx_next_to_use variable, and increment the variable. The context descriptor refers to the following data TX descriptors. The ip_summed field in the sk_buff structure says if the frame need to be checksummed. The gso_size field of the skb_shared_info structure associated with the sk_buff says if the frame need TCP Segmentation Offload.
- Maps all the frame fragments into DMA-capable physical memory (DMA mapping). The mapping is necessary to tell the kernel that we are going to access the memory regions with DMA. The mapping function returns the physical addresses corresponding to a fragment, and doesn't perform a copy. An entry of the tx_buffer_info array is used for each fragment to store its physical address and length. A pointer to the input sk_buff is stored in the entry corresponding to the last fragment. Being the array parallel to the TX ring, the driver starts to use the entry pointed by the tx_next_to_use variable (but don't increment the variable itself, since it will be incremented in the next step).
- Inserts in the TX ring a data TX descriptor for each frame fragment, using the information just stored in the parallel array, and incrementing tx_next_to_use by the number of fragments.
- Updates the TDT register with the tx_next_to_use content, in order to commit the new frame to the hardware.

From the previous list follows that the driver needs to do some per-frame cleanup operations after the hardware has actually sent the frame. This is the reason why TX interrupts are enabled. Therefore, after the hardware has sent one or more frames, it eventually raises an interrupt.

Since the e1000 driver uses the NAPI, the interrupt routine disables the e1000 interrupts and schedules the NAPI context. The e1000 NAPI polling function,

e1000_clean, is (also) in charge of doing the TX cleanup work. This work is done invoking the function e1000_clean_tx_irq.
Starting from the entry pointed by tx_next_to_clean, the polling function fetches the TX descriptors to see what descriptors have the DD bit set, and so have been processed (used) by the hardware. For each used descriptor, it releases the TX resources using the information stored in the corresponding entry in the parallel tx_buffer_info array. In more detail, the driver undoes the DMA mapping of each data TX descriptor and frees the sk_buff of the last data TX descriptor associated with each frame sent.

2.5.4 RX operation

In order to access the RDT register only when necessary, the driver has a shadow variable, rx_next_to_use, that is intended to be used in place of the RDT when possible. The RDT register is updated with the variable content only when the driver wants to give new memory buffers to the hardware. In this way is never necessary to read the RDT register, since one can read the shadow variable instead. Similarly, there is a shadow variable, rx_next_to_clean, also for the RDH register. The shadow variable is incremented once for each used RX descriptor that has the DD bit set (see below). In this way it's never never necessary to read the RDH register, since one can read the shadow variable instead[24].
Once again, the shadow variables are extremely useful to minimize the number of VMExits.

Similarly to what happens for the TX operation, the driver mantains an array parallel to the RX ring, the rx_buffer_info array, in order to keep trace of per-descriptor information that can be used to release the resources when the reception is completed. As an example, each entry in this parallel array contains the DMA-mapped physical address of a memory buffer used by the adapter to receive a frame. The driver must undo the DMA-mapping, but only after the memory buffer has been used by the hardware.

When initializing the the resources for RX operation, the e1000 ndo_open method also calls the e1000_alloc_rx_buffers() function passing $N-1$ as cleaned_count input parameter, so that this function allocates $N-1$ memory buffers to be used for frame reception (see section 2.3.1). In this way the adapter can accept incoming frames as soon as the reception is enabled in the hardware.

In more detail, for each buffer to allocate, the e1000_alloc_rx_buffers function:

1. Allocates a new sk_buff structure that has enough data room to store a maximum size ethernet frame.

[24]Recall that the driver normally does not need to write the RDH register, except for register initialization.

2. Stores a pointer to the `sk_buff` into the entry of the `rx_buffer_info` array indexed by the `rx_next_to_use` content. In this way this pointer can be passed to the network stack by the RX interrupt routine after the memory buffer contained into the `sk_buff` itself is used by the hardware.

3. Maps the memory buffer so that it can be accessed in DMA, similarly to what happens for TX frames (see section 2.5.3). The physical address of the memory buffer is stored in the same `rx_buffer_info` entry.

4. The buffer physical address is also written into the RX descriptor corresponding to the `rx_buffer_info` entry[25].

5. The `rx_next_to_use` variable is incremented. If is the case (see below) the RDT register is then updated with the `rx_next_to_use` content.

A write to the RDT register is performed every 16 allocated memory buffers, and at end of the `e1000_alloc_rx_buffers` function.
There is a tradeoff here: Writing the RDT at each iteration would be too slow, and writing the RDT only at the end of the function would cause the receive side to be poorly reactive.

After one or more new frames are received and stored in the physical memory, an interrupt is eventually raised. The interrupt routine disables the e1000 interrupt and schedules the NAPI context (see 2.5.3). The e1000 polling function is (also) in charge of doing the RX work.
Though the e1000 adapters are able to receive also *jumbo* frames[26], in the following we will describe only what happens when conventional ethernet frame are used[27].
In order to do the RX work, the e1000 polling function invokes the `e1000_clean_rx_irq` function. This function starts to fetch and clean used RX descriptors beginning from the one indexed by `rx_next_to_clean`. It keeps working while the NAPI budget is not exhausted and the descriptors have the DD bit is set. The latter indicates that the RX descriptor has been used by the hardware to receive a frame.

For each RX descriptor to clean:

1. The used memory buffer is DMA-unmapped. The buffer physical address is taken from the `rx_buffer_info` entry corresponding to the RX descriptor.

2. If the received frame is shorter than 256 byte[28], the copybreak mechanism fires: A new minimum size `sk_buff` structure is allocated and the frame data are copied, while the old `sk_buff` will be recycled. This should improve packet loss for small packets, because the socket receive queues get full earlier if the enqueued `sk_buff` are bigger.

[25] Recall that the RX ring and the `rx_buffer_info` arrays are parallel
[26] ethernet frames that can carry up to 9000 bytes of payload
[27] Anyway, the jumbo frame handling it's not very different.
[28] This number is a kernel parameter that can be tuned by the user.

3. Some fields of the `sk_buff` structure built around the received frame are properly initialized[29] and the frame is handed off to the network stack (see the `e1000_receive_skb()`) function. A pointer to that structure is taken from the same `rx_buffer_info` entry, if the copybreak mechanism did not fire.

[29] For example the `ip_summed` and the `protocol` fields.

Chapter 3

Optimizations of emulated e1000 performance

In chapter 2 we have given an overview of the software environment we will deal with in this work. In this chapter we will analyze problems and bottlenecks of the current system implementation and propose solutions that are able to remove them.

First of all we have to point out what we are going to optimize. We consider two scenarios.
In the first scenario, *guest to host* (G2H), a VM is connected to the host through a host-only virtual LAN (see section 2.2.3). The VM runs an application that sends UDP packets of a given size at a given average rate, while the host runs an application that receives all the UDP packets it can. The second scenario, *host to guest* (H2G), is very similar to the first, but the host and guest roles are swapped: the host runs the UDP sender while the guest runs the UDP receiver. These scenarios are depicted in figures 3.1 and 3.2.
Our goal is to maximize the packet rate of the application running on the VM, both on the transmit and the receive side.

In our experiments, we will try to give to each guest one or two VCPU, and discuss the differences between the two cases.

3.1 Analysis of the existing implementation

We are now going to show and discuss the performance of the existing solution, e.g. of the machanisms illustrated in section 2.4 and 2.5. All the measurements have been done on a laptop machine with an i5-2450M processor, which is endowed with four cores running at 2.5GHz[1].

[1]The linux CPU frequency manager has been used to set the CPU frequency at the maximum frequency.

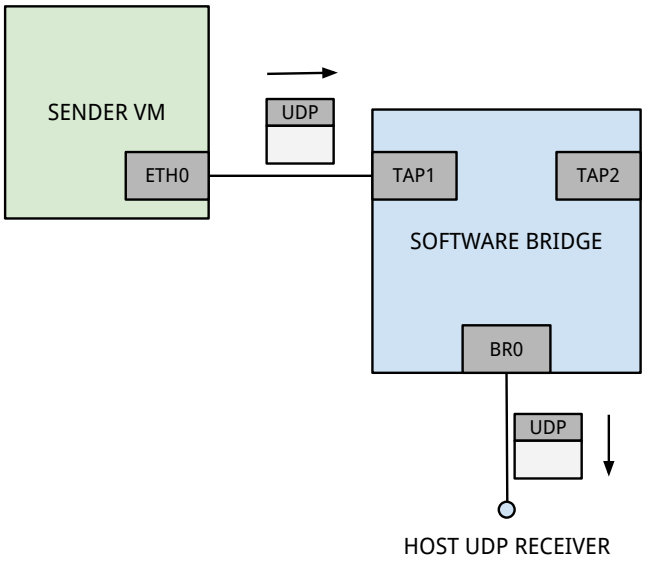

Figure 3.1: Communication scenario in which and UDP sender on a VM sends packets to an UDP receiver on the host. The VM and the host communicate through a virtual LAN made of TAP devices and an in-kernel software bridge.

3.1.1 TX performance

In this experiment a VM runs an UDP sender that sends UDP packets to an UDP receiver that runs on the host. The sender is just an infinite loop where each iteration invokes the `send` system call. The buffer passed to the system call is always the same and its length is such that a minimum size ethernet frame (60 bytes) is sent on the wire for each `send`. With this test we push to the limit the TX performance in the short packet case.

1 VCPU test

The measurement results are shown in the table 3.1. The guest has been assigned 1 VCPU, and all the values are computed counting the number of occurrences of each event over a 1 second period and then dividing the count by the period itself. As we can see, the TX rate is very modest, about 20.6 Kpps. There is a TX notification (a write to the TDT register) and a TX interrupt for each packet sent. The total number of MMIO accesses is 6 times the number of interrupts.
The high MMIO access rate is due to the interrupt routine. When invoked, in fact, the interrupt routine has to VMExit at least 5 times:

1. Read the ICR (Interrupt Cause Read) register, in order to get the Interrupt reason (if any).

Existing implementation	**1-VCPU**	**2-VCPUs**	
Interrupt rate	20.6	23.6	KHz
TX packet rate	20.6	30.4	Kpps
TX bitrate	15.0	22.1	Mbps
TX notifications rate	20.6	30.4	KHz
MMIO write rate	61.7	61.9	KHz
MMIO read rate	61.7	47.3	KHz

With interrupt moderation	**1-VCPU**	**2-VCPUs**	
Interrupt rate	3.8	3.8	KHz
TX packet rate	47.8	47.2	Kpps
TX bitrate	34.8	34.3	Mbps
TX notifications rate	47.8	47.2	KHz
MMIO write rate	55.5	54.8	KHz
MMIO read rate	11.5	11.5	KHz

With interrupt moderation and TDT write batching	**1-VCPU**	**2-VCPUs**	
Interrupt rate	1.8	2.8	KHz
TX packet rate	163.5	145.0	Kpps
TX bitrate	119.0	105.6	Mbps
TX notifications rate	1.8	2.8	KHz
MMIO write rate	5.5	8.3	KHz
MMIO read rate	5.6	8.4	KHz

With TDT write batching (no interrupt moderation)	**1-VCPU**	**2-VCPUs**	
Interrupt rate		7.8	KHz
TX packet rate		95.4	Kpps
TX bitrate		69.5	Mbps
TX notifications rate		7.8	KHz
MMIO write rate		23.5	KHz
MMIO read rate		23.5	KHz

Table 3.1: Guest to host statistics. These four tables show some statistics about a sender VM in various situations (original implementation, moderation patch, TDT-write-batching patch and combinations). Each table reports a set of measurements relative to a 1-VCPU VM and another set relative to a 2-VCPUs VM.

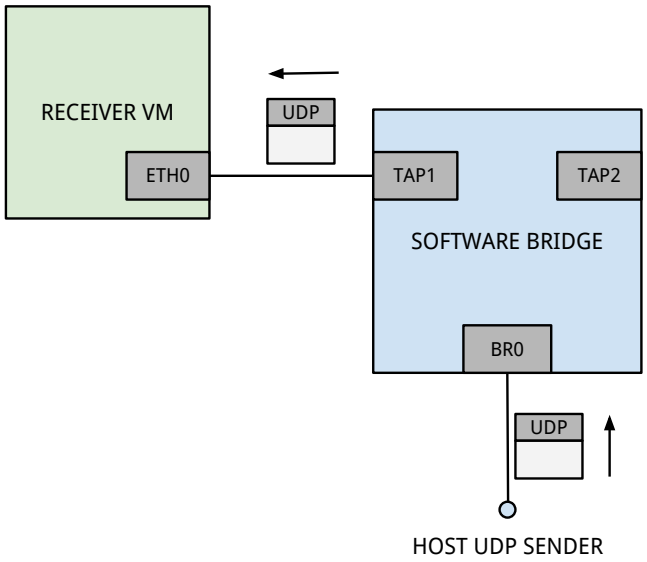

Figure 3.2: Communication scenario in which and UDP sender on the host sends packets to an UDP receiver on the VM.

2. Write to the IMC (Interrupt Mask Clear) register, in order to disable the e1000 interrupts.
3. Read from the STATUS register, in order to flush the previous register write.
4. After the NAPI work is completed, write to the IMS (Interrupt Mask Set) register in order to enable the interrupts.
5. Read form the STATUS register, in order to flush the previous register write.

The sixth MMIO access per interrupt is due to the TX notification.

Even if the hardware interrupt moderation is not emulated, the Linux e1000 driver uses the NAPI interface (section 2.5.1), so a software interrupt moderation is actually implemented. Unfortunately, it doesn't work for TX interrupts, and the reason for this is actually very simple. In section 2.4.1 we have seen that the TX emulation is done in a synchronous way, that is the VCPU thread that writes the to TDT register, after the VMExit, executes the e1000 frontend, the TAP backend and raises the interrupt pin. After that it does a VMEnter coming back to the guest world, but with a TX interrupt to handle. Since the guest has only a VCPU, the VCPU must be used to handle the interrupt, and cannot be used to insert more TX frames into the ring (e.g. to call the `ndo_start_xmit` method). Consequently, the NAPI polling function will only clean the TX descriptor(s) associated to the frame that has just been sent. In conclusion, if we write to the TDT register every time the

`ndo_start_xmit` method is invoked, we are doomed to receive an interrupt for each frame we send, and so to have low performance.

2 VCPUs test

The same experiment has been done assigning 2 VCPU to the guest. The measurement results are shown in the table 3.1.
Compared with the 1-VCPU case, the situation is slightly better, but still modest. Here we have about 30 Kpps, and an interrupt rate that is significantly less than that. This means that the TX interrupt routine (e.g. the polling function), on average, handles more than one frame. Note that there is still only one TX frame sent for each TX notification, like in the 1-VCPU case. The improvement is due to the higher degree of parallelism in the guest, that allows the NAPI to coalesce some interrupts, so that on average about 1.29 data TX descriptors are cleaned for each interrupt. This happens because while one VCPU is executing the interrupt routine (or the NAPI context is active) with the e1000 interrupts disabled, the other VCPU can find the time to insert another TX frame in the ring, execute the frontend and backend and return to the guest without raising an interrupt because they are disabled. This is why, sometimes, the NAPI polling function cleans more than one data TX descriptor.

Discussion

The low performance is basically due to two problems: The hardware interrupt moderation is not emulated in QEMU and there is a TX notification for each frame to send. This is true in both 1-VPCU and 2-VCPU cases.
If we coalesced the interrupts, we could amortize the 5 VMExits and the interrupt overhead over more TX frames. Implementing the emulation of the hardware interrupt is then then an optimization we can implement.
If we had a way to coalesce the TX notifications we could amortize the cost of a notification over more TX frames. This a second optimization we can implement.

We've not discussed the performance with bigger packet sizes, because the problems involved are exactly the same. When the packets are big there is more work to do because the packet copies are more expensive, and so the performance are lower. However, the optimization are very effective also for the big packet case, because the overhead due to the copies is significantly lower than the overheads due to TX notifications and interrupts.

3.1.2 RX performance

In this experiment a VM runs an UDP receiver that receives UDP packets from an UDP sender that runs on the host. The receiver is just an infinite loop where each iteration invokes the `recvmsg` system call. No processing is made on the received

packet. The UDP sender is similar to the sender used in section 3.1.1, but after each `send` system call the process does some busy waiting in order to send UDP packets at a given rate. With this test we push to the limit the guest RX performance in the short packet case.

The receiving process is generally more problematic than the transmit process, because of the *livelock* problem (see section 2.3.2). When the incoming packet rate and/or the interrupt rate are too high, or the traffic is very irregular with high traffic peaks, the guest OS (device driver and network stack) does a lot of processing before trying to put the packet in a socket receive queue, but in the end is forced to throw them away because the queue is full. In its turn, the queue is full because the receiver process doesn't have enough time to read all the packets from the queue.

For this reason we introduce the concept of (receiver) *critical* rate. We define the critical rate as the incoming packet rate that, if exceeded, cause the receiver VM to enter a unstable state and/or a state characterized by bad performance. In more detail, in our experiment we will see the RX rate measured by the UDP receiver as a function of the RX rate measured by the network adapter (the incoming rate). When the incoming rate is low enough, the UDP receiver will measure the same rate, because no packets are dropped. As the incoming rate gets higher a little percentage of packets starts to be dropped, and so the UDP-measured rate is a little lower than than the incoming rate, but still increases when the incoming rate increases. When the incoming rate exceeds the critical rate, however, the UDP-measured rate starts to decrease when the incoming rate increases. Moreover, beyond the critical rate the UDP-measured rate generally oscillates, e.g. the receive system becomes unstable and its responsiveness becomes very low. Figure 3.3 shows an example of the function we are talking about.

The critical rate is very significant, since it tells what is the maximum incoming rate that a guest OS running simple receiver process can accept while being in a stable state.

1 VCPU test

In this test one VPCU is assigned to the guest. The measured critical RX packet rate is about 14.4 Kpps. The measurement results shown in table 3.2 are taken when the received rate is about the same as the critical rate.
The critical RX rate is very low. If we keep incrementing the incoming RX rate beyond the critical point, the system enters a livelock state. Since more RX interrupt work is requested and we only have a VCPU, the receiver user process has less CPU time than before, even if there are more packets to receive. As a result, the RX throughput seen by the user process drops immediately after the critical rate, the system become extremely instable, and most of the packets are dropped. With packet rates higher than 15 Kpps the system becomes unusable.

Existing implementation	**1-VCPU**	**2-VCPUs**	
Interrupt rate	14.3	6.7	KHz
RX packet rate	14.4	185.7	Kpps
RX bitrate	10.5	135.1	Mbps
RX notifications	14.3	13.7	Mbps
MMIO write rate	42.9	22.7	KHz
MMIO read rate	42.9	13.5	KHz

With interrupt moderation	**1-VCPU**	**2-VCPUs**	
Interrupt rate	3.8	2.1	KHz
RX packet rate	137.1	216.7	Kpps
RX bitrate	99.8	157.7	Mbps
RX notifications	10.3	14.3	Mbps
MMIO write rate	18.0	18.5	KHz
MMIO read rate	11.5	6.3	KHz

Table 3.2: Host to guest statistics. These two tables show some statistics about a receiver VM in two different situations (original implementation and moderation patch). Each table reports a set of measurements relative to a 1-VCPU VM and another set relative to a 2-VCPUs VM.

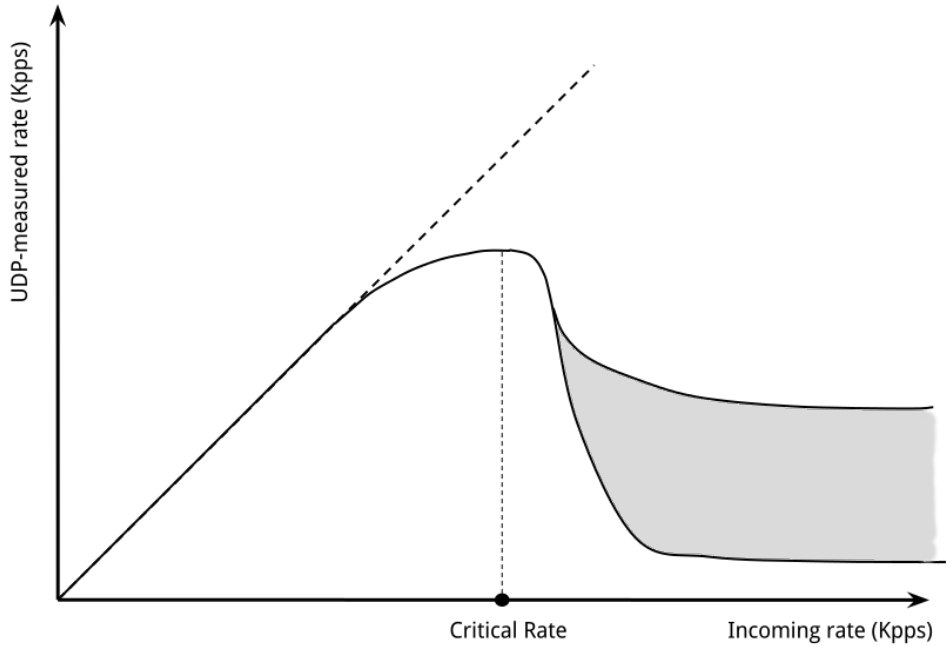

Figure 3.3: This plot is an example of RX rate measured by the UDP receiver as a function of the RX rate measured by the network adapter (incoming rate). The critical rate is the incoming rate for which this function has a maximum. The gray area represents the oscillation (upper and lower bound) of the UDP-measured RX rate.

Let's analyze what happens when the livelock doesn't shows up (table 3.2) The interrupt rate is a little lower than the RX packet rate, and this means that some interrupts have been coalesced by the NAPI. This is possible even with a single VCPU, differently from what happens int the TX case, because (see section 2.4.2) the hardware emulation (backend and frontend) is done by the IOThread, while the guest is executed by a VCPU thread. Parallelism is therefore possible, because the IOThread could insert a new frame in the RX ring while the NAPI is polling, and/or the interrupt could be delayed because the interrupts are disabled.
However, the two rates are almost the same, and so we still have approximately an interrupt for each frame received, which means that the NAPI software moderation isn't not working well. More precisely, we have measured the distribution of the amount of RX NAPI work done each time the polling function is called and reported it in table 3.3.

Why is the NAPI working so bad? The problem here isn't related to lack of parallelism, like in the TX case, but is due to the guest being too fast. In fact, we can observe that in this experiment the overall system is composed of a *producer*

NAPI RX work	Percentage
0	0.05%
1	99.485%
2	0.335%
≥ 3	0.13%

Table 3.3: Host to guest NAPI distribution with 1 VCPU per guest. The NAPI work is the number of frames handled by the execution of the polling function.

thread, e.g. the IOTHread, and a *consumer* thread, e.g. the VCPU thread. The producer is basically an infinite loop that in each iteration gets a frame from the TAP (waiting/sleeping if no frames are ready to be read), inserts it in the RX ring and raises an interrupt if enabled. The consumer is woken up by an interrupt and schedules the NAPI context. The NAPI polling functions is a loop that on each iteration extracts an RX frame from the ring and push it to the stack, until there is no work left.
If the consumer iteration is on average slower than the producer iteration, the latter is very likely to find the interrupts disabled after inserting a new RX frame, and the consumer is very likely to see the new frame while is executing the polling function corresponding to a previous interrupt. In this scenario the consumer will (almost) always find work to do, and so the interrupt will (almost) always be disabled, and consequently the NAPI mitigation will work.
On the other hand, if the consumer iteration is on average faster than the producer iteration, the latter is very likely to find the interrupt enabled after inserting a new RX frame, and so an interrupt is raised for each received frame. This is exactly what happens in our experiment. It's important to point out that when there is no more work to do, the NAPI polling function is forced to complete and enable the interrupts, because it doesn't know when the next RX frame is going to come.
Observe that in this experiment the consumer is very slow simply because we have forced the UDP sender to send to a constant packet rate of 14 Kpps, which makes it slow by definition. We cannot go beyond 14Kpps because of the livelock.

Moreover, as we can see in the table 3.2, we also have approximately a write to the RDT register (RX notification) for each interrupt, which is not good. This is also a consequence of the NAPI misbehaving.
Similarly to the 1-VCPU TX case, here we have 6 MMIO accesses for each interrupt. Five of them are exactly the same listed in section 3.1.1, while the sixth one correspond to the RX notification.

2 VCPUs test

The same experiment has been done assigning 2 VCPU to the guest. Here the critical rate is way higher, about 175 Kpps. Moreover, if the rate is passed, the system still works, even if packets start to be dropped and it becomes unstable (the dropping percentage varies very much between 5% and 85%). The performance doesn't drop immediately after the critical rate, like in the 1-VCPU case: The system is still usable (but not stable) if the incoming RX rate is about 300K.
The measurement results are shown in the table 3.2 when the incoming packet rate is about 185 Kpps.
As we can see, here the NAPI works very well, since the producer here is way faster than 14 Kpps (see section 3.1.2). On average we serve about 27 RX frames per interrupt, so that the high interrupt overhead and the MMIO accesses are amortized. The RX notification rate is about twice as the interrupt rate, because when the polling function handles more frames, it gives new memory buffers to the hardware doing a write to the RDT register every 16 frames handled (see section 2.5.4) and a write at the end of the polling function. Since on average 27 frames are served, we have on average $\lfloor \frac{27}{16} \rfloor + 1 = 2$ RDT writes.

Discussion

The low performance in the 1-VCPU case is basically due to two problems: The hardware interrupt moderation is not emulated in QEMU and there is a RX notification for each frame to send. The interrupt moderation here is necessary, because the NAPI mitigation doesn't work (because of the livelock).
In the 2-VCPU case the mitigation can still be useful to remove the existing fluctuations in the interrupt rate which cause performance drops, and in general would be useful for the system to be more stable. In our experiment, in fact, we noted that the performance drops when the interrupt rate has a peak (10 KHz).
The second problem has minor negative effects on performance in the 2-VPCU case, because an RX notification is done every 16 frames, so the associated cost is amortized.

We've not discussed the performance with bigger packet sizes, for the same reasons explained in 3.1.1.

3.2 Implementing interrupt moderation

In section 3.1.1 we have seen that we can improve both the RX and TX packet rate if we emulate the e1000 mitigation. A precise emulation is not necessary nor possible, and it's therefore convenient to choose a simple and efficient one. In this work we have implemented the ITR register, the TADV register and the RADV

register. We are not interested in the TIDV and RDTR registers[2].

In our implementation we use a single QEMUTimer (see section 2.2.1), even if the hardware has multiple timers, and so we have to aggregate the meanings and functionalities of the ITR, TADV and RADV registers. We therefore consider the moderation delay to be the minimum delay among the ones specified through the ITR, TADV and RADV registers. When computing the minimum, each register is considered only if its content is valid and if there is a pending event of the proper type. According to the e1000 specification ([7]), a zero value means that the register content is not valid. If there is no pending TX event the TADV content is not considered, while if there is no pending RX event the RADV register is not considered.
Moreover, only the interrupts due to transmission (`start_xmit()` function) and reception (`e1000_receive()` function) are considered. The other interrupts are extremely rare, so they are not intersting.

Let's see how moderation is implemented. We have modified the code so that the frontend calls the `mit_set_ics()` function instead of the `set_ics()` function when it wants to issue an interrupt. When called with the moderation timer (`mit_timer`) inactive, the `mit_set_ics()` arms the timer with the moderation delay and issues an interrupt. When called with the moderation timer active (e.g. the timer has not expired yet), it only accumulates the interrupt cause, but don't issue an interrupt. When the timer expires, an interrupt is sent, and the timer is rearmed only if there is a pending accumulated interrupt cause and the moderation delay is not zero. Every time the timer is (re)armed the moderation delay is computed again, because the mitigation registers content can change, and the pending events could be only RX or only TX events.
In this way we are sure that the minimum inter-interrupt interval is always greater or equal than the moderation delay.

The proposed moderation patch adds about 50 lines of code to the e1000 frontend (hw/e1000.c).

In the following sections we will repeat the same experiments presented in section 3.1 in order to see the improvements obtained with the the moderation patch. With this experiment the Linux e1000 module has been loaded specifying the following parameters:

[2]However, the RDTR register has been added to the frontend only to validate the RADV content. The e1000 speicification, in fact, says that the TADV register is not valid if RDTR contains 0.

Parameter	Value
TxIntDelay	0
TxAbsIntDelay	0
InterruptThrottleRate	4000

In this way we only use the new moderation mechanism (e.g. ITR). It's not necessary to specify the parameters relative to RDTR and RADV, since RDTR is 0 by default (so both RDTR and RADV are disabled). The InterruptThrottleRate parameter actually specifies the Maximum Allowed Interrupt Rate (MAIR), which is inversely proportional to the value that the driver has to put in the ITR register in order to limit the interrupt rate.

3.2.1 TX performance

The results in the 1-VCPU case is shown in table 3.1.
As we can see from the table, there is an important improvement. The interrupt rate is less than 4 KHz, and this is what we expected to see since we have set the MAIR to 4000. With reference to the existing implementation (section 3.1.1, we have a performance gain of 2.32, with about 47 Kpps. Since the interrupts are coalesced by the interrupt moderation mechanism, the NAPI polling function cleans on average 12.58 TX data descriptors each time is called. Remember that the TX emulation is synchronous with the guest VPCU (see section3.1.1), and so the NAPI software moderation it's useless by itself.
Despite of the good improvements, we still have a TX notification for each frame sent, and the performance is limited by this problem.

In order to understand the mitigation effects on performance, we have tried different MAIR values, computing the average TX packet rate on a very long time window (some minutes). The results are shown in figure 3.4. From this plot we can see that the lower the MAIR is, the higher the TX packet rate is. However we cannot set MAIR to a value too low, because this would increment the latency too much. In fact while the moderation timer is on, we don't let the emulated hardware raise any interrupt, and so the responsiveness is limited by the moderation delay, namely by the ITR. As stated previously, in this study we want to maximize the packet rate, and we are not interested in minimizing the latency.

The measured results in the 2-VCPU case are shown in table 3.1.
The results are similar to the 1-VCPU case, because the incremented parallelism is not exploited. Even if there is parallelism between the interrupt routine and the transmission path, the TX clean work is not very expensive. Therefore we don't benefit from a second VCPU, or the little benefits are compensated by the overhead involved in the SMP management (e.g. locks and barriers).

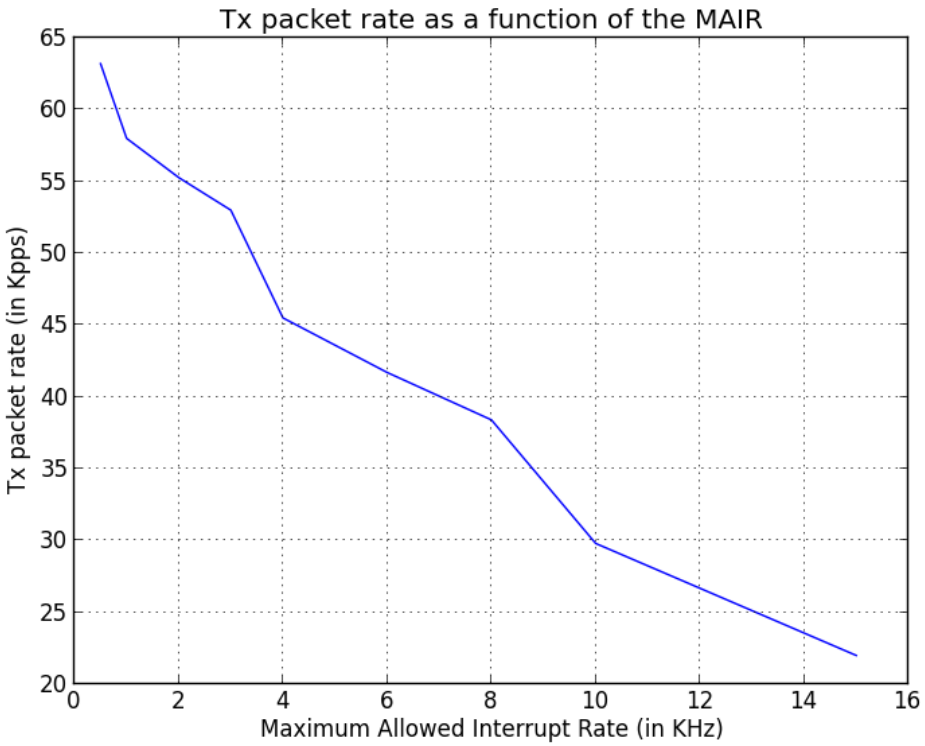

Figure 3.4: Measured TX rate as a function of the Maximum Allowed Interrupt Rate (MAIR). As we can see, the TX rate increases as the MAIR descreases. However, we cannot decrease the MAIR too much, because the latency worsens as the MAIR decreases.

3.2.2 RX performance

The measured critical rate with 1-VCPU is about 150 Kpps. The table 3.2 shows the results obtained when the incoming RX rate is about 137 Kpps.
As we can see, there is a huge improvement in the packet rate performance, because on average we amortize the interrupt related overhead over about 35 frames. Since the interrupt rate is lower, also the RX notification rate is lower. On average we should expect $\lfloor \frac{35}{16} \rfloor + 1 = 3$ RDT writes for each interrupt (see section 2.5.4), and so a notification rate of about $3 \cdot 3.838$ KHz = 11.514 KHz, which is similar to the measured one (10.3 KHz).
If we increase the incoming RX rate, the performance of the UDP receiver gradually degrades, but we don't run into a complete livelock (like the livelock we have seen in section 3.1.2), because the interrupt rate is bounded.

In order to understand the mitigation effects on performance, we have tried differ-

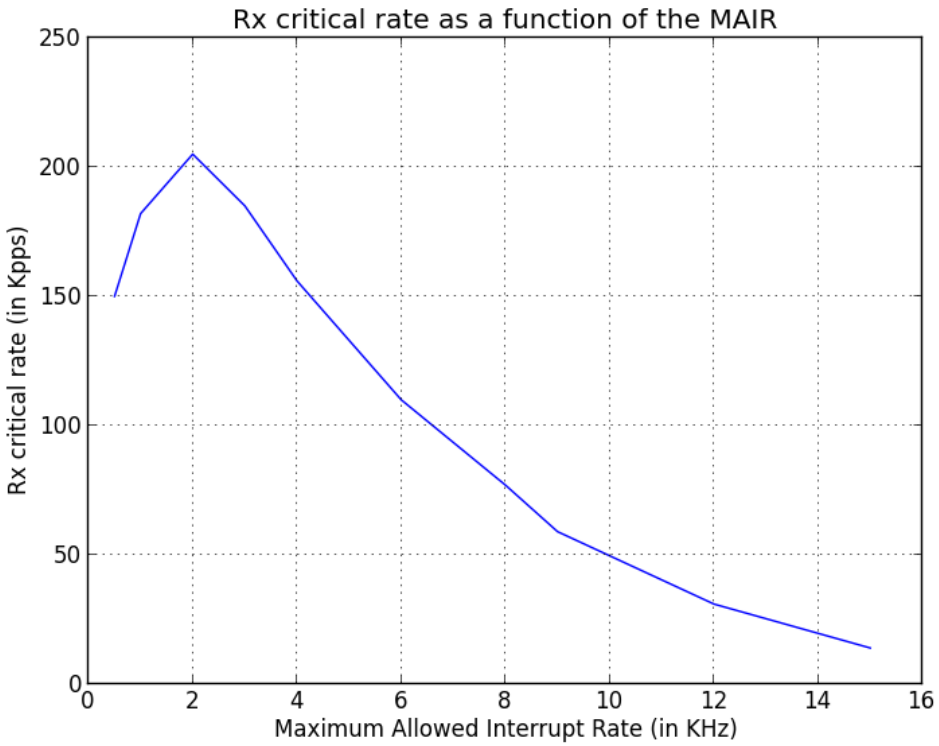

Figure 3.5: Measured critical rate as a function of the maximum allowed Interrupt Rate (MAIR). If the MAIR is too high, performance worsens because the interrupt rate is too high. If the MAIR is too low, performance worsens because the RX ring gets full and guest is not notified immediately.

ent MAIR values, measuring the critical rate for each value. The measurements are shown in figure 3.5. The plot shows that if we increase the MAIR in in the region [2 KHz, $+\infty$], the performance gradually decreases, since we are less and less restrictive on the Maximum Allowed Interrupt Rate. On the other end, if we decrease the MAIR below 1.5 KHz, the throughput starts to decrease, because the RX ring gets full and the guest is not notified in a timely manner. Moreover, we cannot choose the MAIR to be too low because of the latency (see section 3.2.1);

The 2-VCPU case results (MAIR = 4 KHz) are shown in table 3.2
Similarly to what happens without moderation, a second VCPU improves the RX performance, because the improved parallelism makes the NAPI moderation to play an active role. The measured critical rate is about 215 Kpps (+37% w.r.t. 1-VCPU case). We can see that the NAPI mitigation is effective observing that the average interrupt rate is half part of the MAIR, and so the MAIR is not restrictive at operating speed. On average we have about 103 RX frames served for each

interrput, which a very good result. The expected average RX notification rate is $(\lfloor \frac{103}{16} \rfloor + 1) \cdot 2.098$ Khz = 14.686 KHz, which is very similar to the measured result (14.321 KHz).

3.3 Implementing TDT write batching

In section 3.2 we have seen how RX/TX packet rate performance can be improved with a minimal patch to the e1000 frontend that implements an interrupt moderation mechanism. However, the TX notification problem still exists and limits TX performance. The interrupt moderation mechanism cannot help to mitigate this problem. For these reasons, we propose a TDT *write batching* algorithm, which will be explained in the following section.

3.3.1 Implementation

A simple approach would be to coalesce TX notification using a counter variable. The counter is initialized to zero and is incremented every time the `ndo_start_xmit` method is invoked. A TDT write is done to notify the pending transmission only when the counter reaches a threshold, e.g. 20. Each time we notify we also set the counter to 0. In this way we are able to reduce the TX notification rate by 20 times. Unfortunately this approach would stop working when the guest stop transmitting, because we don't have a way to timely notify pending transmits, if any. Moreover, the `ndo_xmit_method` cannot know when it will be invoked again in the future. For these reasons we should use a kernel timer in order to notify pending transmit after a period of TX inactivity. This timer should be (re)armed every time we don't notify, and so 19 times over 20. This is quite expensive. Moreover, the user should have to choose the threshold value and the timer delay value, and this would add complexity and discourage the users.
In order to get to a zero-configuration and simple implementation we have adopted an other approach.

The idea is to use the interrupt itself as a timer mechanism, and thus the interrupt routine as a timer callback. We use a variable, `bat_pending`, which is 1 to indicate that there is a pending interrupt that will come soon, and is 0 when there is not a pending interrupt. The variable is initialized to 0. When the `ndo_xmit_method` is called for the first time, `bat_pending` is 0: The notification is done and `bat_pending` is set to 1. If the `ndo_xmit_method` is called again before the TX interrupt (due to the previous notification) comes, `bat_pending` is 1: In this case we don't notify, and consequently that transmission become pending. When the interrupt comes, at the begin of the NAPI polling function (the interrupt are disabled), if there are pending transmissions we do a notification, but we don't set `bat_pending` to 0, because the notification we have done will cause another interrupt. If there are no pending notifications we set `bat_pending` to 0, because we don't know when the

next interrupt is going to be, and so we want the next transmission to cause a TX notification.
Note that this algorithm works well only if there is time for the guest to call the ndo_xmit_method before the an interrupt comes. For instance, this can happen if interrupt moderation is implemented, since TX interrupts are delayed.

The proposed patch includes a few other implementation details that are not very interesting. For instance, we have to hold a spinlock while writing to the TDT register and updating the other variables related to the batching mechanism, because we perform this write accesses both in the ndo_start_xmit method and in the NAPI polling function, and so we need mutual exclusion. In addition to that, when we have to check if there are pending transmissions, we should read from the TDT register, but this would be counter-productive. Therefore we mantain a shadow variable (bat_software_tdt) that is always synchronized with the TDT content. Finally, when writing to the TDT register from the interrupt routine we cannot read from tx_next_to_use, since its value is modified without the spinlock held. The bat_shadow_ntu is used to take coherent snapshots of the tx_next_to_use variable. The batching patch adds about 35 lines of code to the e1000 driver. The user can enable or disable the batching mechanism writing 0 or 1 to the batching module parameter.

3.3.2 Improvement analysis

In this section we will repeat the same experiments presented in section 3.2.1 in order to see the improvements obtained with the the batching patch and the moderation patch. The Linux e1000 module has been loaded specifying the same parameters plus the batching parameter, which has been set to 1.

1-VCPU test

The results in the 1-VCPU case is shown in table 3.1.
As we can see there is a big improvement in the TX rate with respect to the previous solutions (existing implementation with or without the moderation patch), because now we have only one TX notification for each interrupt. At this point we have moderated both interrupts and TX notifications and there is nothing else to mitigate. However, the TX path still does not work at his best, because the TX emulation is still demanded to the VCPU thread that executes a TX notification (section 2.4.1). In other words, the processing is synchronous, while in the RX path the emulation can go parallel with the guest.

The TDT write batching algorithm would be intended to process, for each notification, a batch of frame. Unfortunately, this is not exactly true, because of the following factors:

1. The TX processing is synchronous, while it would be better doing it in the IOThread.

2. The TX processing holds the `iothread lock`, preventing other VCPUs to execute emulation code while the lock is held.

3. The TX processing always processes all the TX pending descriptors.

4. The moderation timer works with the host time, so it does not stop running down when a VCPU is not executing guest code, but it is executing emulation code. Therefore a 1-VCPU guest can see a moderation delay which is shorter that expected or even close to zero.

For these reasons, even though the UDP sender never stops sending, the sequence of batch lengths oscillates, expecially in the 1-VCPU case. In the latter case, the sequence of batch notifications alternates batches of length 1 and very big batches (about 200 frames), in a very regular manner. The following batch lengths sequence has been extracted by the emulator while executing with the batching enabled:

... 1, 185, 1, 195, 1, 169, 1, 221, 1, 212, 1, 198, 1, 210, 1, 200, 1, 215, 1, 211, 1 ...

Let's see what happens in more detail. The first time the guest wants to send, the notification is done, `bat_pending` is set to 1, the TX is emulated, an interrupt issued and the moderation timer is armed. When executing the NAPI polling function, there are no pending frames, because the guest has not had time to do anything after the notification, and so `bat_pending` is set to 0. Here the cycle starts.
Now the guest wants to send another frame, the notification is done, `bat_pending` is set to 1, the TX is emulated but no interrupt is issued, because the moderation timer is active. The guest will continue to send other frames, but this time it will find `bat_pending` set to 1 for a while, and can therefore insert many new TX frames in the TX ring without doing any notification. When the moderation timer expires, an interrupt is issued and the timer rearmed. This time the polling function finds a lot of pending frames, and so a notification is done and `bat_pending` is not set to 0. Since there is a lot of TX emulation processing to do, and the moderation timer keep running down, when the processing is finished the timer is very likely to be expired: Therefore an interrupt is issued and the timer is not rearmed, because no events have come in the while. Consequently, when the VCPU enters the guest again (because the processing is done), it has another interrupt[3] to process with no pending TX frames, and so `bat_pending` is set to 0. Here the whole thing starts again.

[3]Here we can see that, from the guest's point of view, the new interrupt has come immediatly after the previous one.

2-VCPU test

The results in the 2-VCPU case is shown in table 3.1.
As we can see, the performance is very good, but slightly inferior if compared with the 1-VCPU case. As pointed out in the 1-VCPU analysis, when a VCPU is doing the TX emulation, the lock is held, and so the other VCPU cannot do any VMexit, limiting the system parallelism, because that VCPU would be blocked on the lock. However, it's not necessary to do a VMExit in order to insert a new TX frame in the ring when `bat_pending` is 1, and so the second VCPU can actually have the time to insert more frames. For these reasons the sequence of batch lengths has an evolution which is different from the 1-VCPU case, but there is still a regular oscillation. The following sequence has been exctracted while running the emulator:

... 67, 15, 87, 17, 77, 16, 86, 16, 85, 16, 87, 16, 69, 16, 87, 16, 87, 16, 87, 16 ...

We can also observe the interrupt rate is higher w.r.t. the 1-VCPU case. The higher interrupt rate can be explained because the batches are shorter. Shorter batches means that the `start_xmit` is shorter on average, and so the QEMU event-loop is more responsive, having more chances to issue an interrupt, either in the timer callback or at the end of the `start_xmit` itself. Therefore an interrupt is less likely to be delayed more than the moderation delay because of a long TX processing. To close the cycle, since interrupt rate is higher, the guest has less time to replenish the TX ring, and so the batches are shorter.

In other words, with 2 VCPUs the TX path converges towards a different stable state, which is incidentally less efficient. This is an anomaly, since with more VCPUs, and so with more computational power, we have less performance.

3.3.3 Batching without interrupt moderation

Even though the batching patch works well when the moderation is implemented, it's interesting to do some test when the moderation is off.

With 1-VCPU guests the patch is completely useless and harmless, because the TX interrupt is never delayed, and the whole TX path is synchronous with the VCPU itself. In more detail, the first TX notification is performed because `bat_pending` is 0 and so `bat_pending` is set to 1. Then the VCPU executes the TX emulation and raises an interrupt, reentering the guest. When the guest executes the polling function, there are no pending TX frames (because the VCPU had no chances to insert new frames) and so `bat_pending` is set to 0. The next time the guest wants to send a frame the same thing happens again. So we have a notification for each frame to send, e.g. the batching patch is useless.

With 2-VCPUs guests the situation is different, because a TX interrupt is issued only at the end of the TX emulation, e.g. when all the notified TX descriptors have

been processed. So with respect to the other VCPU, the TX interrupt is actually delayed. In more detail, the first time the guest wants to send `bat_pending` is 0, so the latter is set to 1 and a notification is done. While one VCPU is doing the TX emulation or is servicing the TX interrupt, the other VCPU is able to insert more frames in the TX ring, so that the next time the notification is done many descriptors are ready to be processed. In this way the batching strategy is able to amortize the TX notification overhead over many frames. We have run an experiment with 2-VCPU and obtained the results reported in table 3.1.
Here is an extracted sequence of batch lengths

... 8, 16, 9, 15, 8, 15, 8, 15, 7, 15, 8, 15, 7, 16, 7, 15, 8, 7, 6, 16, 8, 14, 7, 16, 8 ...

This result is interesting because the batching patch doesn't require modification to the emulator, but only to the guest device driver. Therefore it can be applied to other emulators that don't implement interrupt moderation.
As an example, we tried to run a similar test under VirtualBox, using two UDP senders on the 2-VCPU guest and a receiver on the host. In this situation the total packet rate is about 168 Kpps with the batching enabled, while if the batching is disabled the total packet rate is about 60 Kpps. We have not used a single UDP sender because it was not enough to trigger the batching mechanism. This is probably due to the different implementation of the e1000 emulation (e.g. a different thread organization).

Chapter 4

A paravirtualized e1000 adapter

In chapter 3 we have proposed two simple patches that boost the e1000 performance. The moderation patch involves only modification to the hypervisor, while the batching patch involves only modification to the guest device driver. These patches can be applied independently on each other, although batching generally works better if used together with moderation.

However, in both cases the we have respected the original e1000 interface specification. In this way the guest can use the e1000 adapter with its original (or patched) driver and be unaware that it is actually in a Virtual Machine environment, and that the e1000 adapter is emulated. This unawareness is the essence of the *full virtualization* concept: The guest doesn't know to be emulated, so that we can run an unmodified OS on top of the VM and everything works fine. Also we can use the patched driver with a real e1000 device, or we can use the patched hypervisor with a different e1000 driver (e.g. a Windows e1000 driver). Whatever combination we make, it will work fine, because we have always respected the e1000 interface.

Another approach that sometimes is used is *paravirtualization*, a general concept that describes situations in which the guest is aware of being in a Virtual Machine environment, and cooperates with the hypervisor in order to make the virtualization simpler and/or to get better performance.

4.1 Device paravirtualization

In this thesis we are interested in device paravirtualization, and in particular in network device paravirtualization. With this kind of paravirtualization, only the *paravirtualized* device driver is aware of the virtualization, while the rest of the guest system is not. We can obtain a paravirtualized driver either by modifying an existing real driver, or by creating a new *fake* driver and a new fake device emulator. In the latter case the driver correspond to a new virtual (fake) device that does not really exist, but is just a stub used to communicate with the hypervisor exporting

to the guest OS the same interface exported by a real driver, so that the guest OS can make use of it without being aware that the device is a fake one. In both cases the paravirtualization requires hypervisor support.
In our case we could modify the e1000 Linux driver and the e1000 QEMU frontend, or we could create a new Linux network driver and a new QEMU frontend.

How can paravirtualization improve performance? Device hardware specifications are generally very complex, and often inlcude a lot of physical details, offloading capabilities and other hardware related features. When emulating the device, most of these details and features are just useless, or is not worth/possible implementing it. Moreover, the devices communicate with the OS mainly through register accesses because register accesses are not expensive in hardware, but they do are expensive within emulators, since they cause VMExits. In other words, emulating real device is complicated and inefficient.

The idea behind paravirtualization is that most of hardware-related details are source of useless overhead, and this overhead can be easily avoided if the driver knows that it is talking to a virtual device and not to a real hardware. As an example, the e1000 driver has to do at least 5 VMExit while executing the interrupt routine, but if we knew that the e1000 device is virtual, some (or all) of these would become useless, or at least could be replaced with something cheaper.
The purpose of a paravirtualized devices is therefore to estabilish an efficient communication between the device driver and the emulator (e.g. the QEMU frontend), while the guest OS thinks of the driver as being the driver of a real device. In order to make the communication efficient, we have to minimize VMExits, and so register accesses and interrupts. The communication should be done, when possible, through shared memory. The TX ring is an example of shared memory used for communication: The driver writes to the TX ring and the frontend reads from it (and then also writes back). The RX ring is another example.

However, also a paravirtualized driver needs to access register, since a register access is the only way the driver can notify the emulator to start some processing or, in general, to have side effects. The difference with a real driver is that a paravirtualized one does a register access only when is really essential, e.g. for notifications. For everything else, the communication is done through shared memory. A real device driver doesn't worry so much about accessing registers.
Similarly, also a paravirtualized device emulator needs to send interrupts, since interrupts are the only way the emulator can notify the driver.

4.2 The Virtio standard

Virtio ([13]) is a virtualization standard that aims at high I/O performance through device paravirtualization. This is done creating a completely new set of device

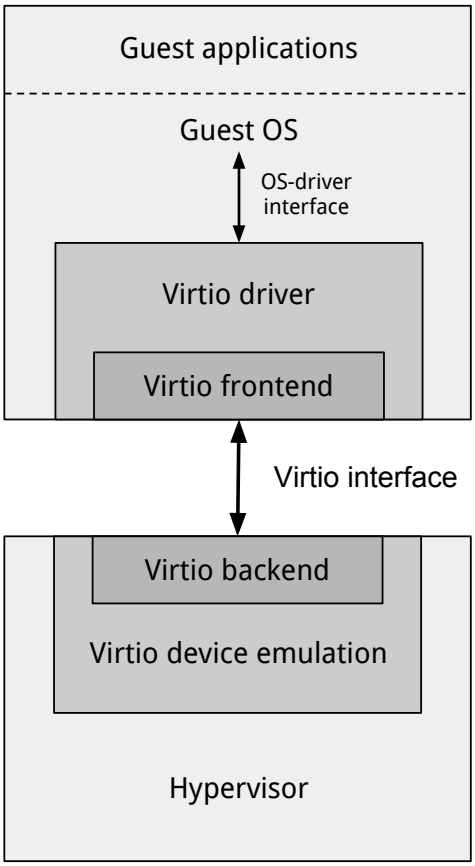

Figure 4.1: A guest and its hypervisor communicate through the Virtio interface. The Virtio backend and Virtio frontend, together, implement the Virtio interface.

drivers, which are able to communicate efficiently with a Virtio-capable hypervisor. Its approach is similar to the Xen I/O paravirtualization ([17]) and the VMware *Guest Tools* ([15, 16]). Virtio is an effort to estabilish a standard interface between drivers and hypervisors for paravirtualized I/O, in order to increase the code reuse across different platforms. In this way we avoid having an independent I/O paravirtualization solution for each hypervisor. The Virtio standard defines different I/O (fake) devices, including a network adapter, a SCSI disk and a serial interface, and is currently supported by QEMU and Virtualbox. Linux and Windows drivers are available for Virtio devices.
As we can see from figure 4.1, the Virtio interface is implemented through a *virtio frontend* in the guest OS, and a *virtio backend* in the hypervisor. All the virtio device drivers can share the virtio frontend code. The task of a virtio driver is therefore to convert the OS representation of the data to exchange (e.g. a network packet) into the standard virtio data format, or the other way around. All the virtio

device emulators (e.g. the QEMU frontend for each virtio device) can share the virtio backend code. The task of a virtio device emulator is therefore to convert the virtio representation of the data to exchange into the hypervisor specific representation (e.g. a buffer containg an ethernet frame), or the other way around. In this way, the guest OS and the hypervisor can communicate through the virtio infrastructure using an efficient and general mechanism.

The organization illustrated in figure 4.1 is actually similar to the one used for real device emulation (e.g. e1000 emulation). However the Virtio interface is explicitely designed for efficient communication between driver and hypervisor, while this is not true for real device hardware/software interfaces.

In order to estabilish an efficient communication channel, the Virtio interface implementation uses MMIO accesses and interrupts only to notify the other peer, and never to exchange data information. Whatever data exchange is done through shared memory.

4.2.1 Virtual queues

Central to the Virtio interface is the *virtual queue* abstraction, a queue that connects a virtio frontend to a virtio backend. A virtual queue is simply a queue into which buffers are posted by the guest driver for consumption by the hypervisor: In this way the two peers can exchange data. A virtual queue can be used to exchange data in both directions: Therefore the posted buffers can be used both for output and for input operations. Drivers can use zero, one or more queues, depending on their needs. For example, the virtio network device uses two virtual queues (one for receive and one for transmit), while the virtio block device uses only one. The buffers used to exchange data are represented in Virtio using scatter-gather lists[1]. With a single operation a virtio frontend can send a scatter-gather list to a virtio backend. A single scatter-gather list can specify both input and output requests. For example, the guest may send to the hypervisor a SG list containing three buffers: An output buffer that specifies a command and two input buffer that will be filled by the hypervisor with the response[2].

In more detail, when the guest wants to make requests to the hypervisor through a virtual queue, it invokes the `add_buf` method on the virtual queue object, passing a SG list and a non-NULL token which is returned when the SG list has been consumed. As we have seen previously, a single SG list can be used to pass many output buffers (e.g. network packets to send) and many input buffers (e.g. memory buffers where the hypervisor can store received network packets). The method returns the amount of space left in the queue, so that the guest can stop adding new

[1] Each element in the list represents the guest physical address and the length of a physical contiguous chunk of memory.

[2] This example could be valid for a virtio disk.

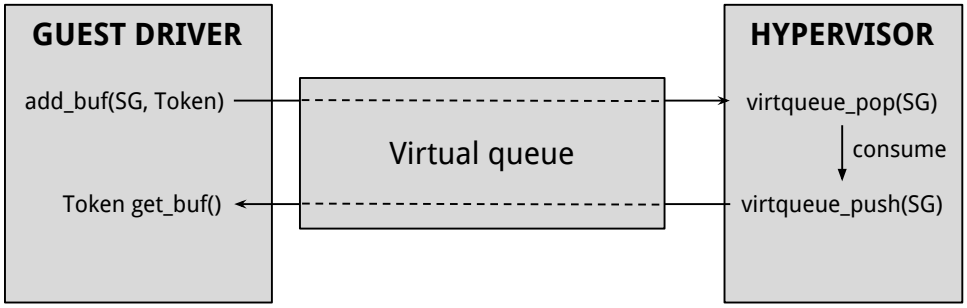

Figure 4.2: Virtual queue operations. The guest adds new buffers (represented as scatter-gather lists) to the virtual queue. The hypervisor consumes (uses) those buffers and returns them to the virtual queue. Tokens are used by the guest to identify buffers.

buffers when the queue is full. The `add_buf` method doesn't notify the hypervisor about the new requests, but only inserts the new buffer in the virtual queue. When a virtio driver wants to notify the hypervisor, it has to call the `kick` method. Of course the driver should kick the hypervisor only when necessary, and try to add as many buffers as possible before *kicking* the hypervisor.
When the hypervisor is notified, it extracts (pops) an SG list from the virtual queue and process the request, maybe asynchronously. When the processing is done, the hypervisor returns the used SG list to the virtual queue. The guest can poll for the request completion through the `get_buf` function. This function is not blocking and returns NULL if there are no returned used SG lists, or returns the token associated to a SG list that has been consumed. Only when `get_buf` is invoked the queue space used by a previous `add_buf` is freed.
The whole process is depicted in figure 4.2.

In order to avoid busy waiting, the driver can provide a callback function to a virtual queue. This callback will be invoked when the hypervisor notifies that new used buffers have been returned. Since hypervisor notifications are generally expensive (in QEMU-KVM they are implemented as interrupts to the guest, so they are extremely expensive), the driver should implement strategies aimed at mitigating the notification rate (e.g. with NAPI). In order to do that, the driver can enable or disable callbacks (e.g. enable/disable interrupts) invoking the `enable_cb` or `disable_cb` methods on the virtual queue. The `disable_cb` is actually only a hint, there is no guarantee that the callback will not still be called right after, since this would require expensive synchronization: It is just an optimization to reduce unnecessary notifications. The callback function can call the `get_buf` so that it can process the used buffers.

In a similar way, the hypervisor can enable/disable guest notifications (kicks)[3], since also this notifications are very expensive. When guest notifications are disabled, the `kick` method has no effect.

4.2.2 The virtio ring transport mechanism

In the current implementation, the virtual queues are implemented as *rings*, similarly to what happens with network adapters (see section 2.3.1). The Virtio interface hides the rings, so that we could use a different implementation as long as both virtio frontend and virtio backend use the same transport mechanism. Since a virtio ring must be a very general and flexible mechanism for transporting buffers, it is also quite complex, or at least more complex than needed for and efficient paravirtualized network device.

A virtio ring consists of three parts: the descriptor array where the guest chains together length/address pairs (taken from a guest-provided SG list), the *available* ring where the guest indicates what descriptors chains are ready for use, and the *used* ring where the host indicates which descriptors chains it has used. Each descriptor contains the guest-physical address of the buffer, its length, an optional `next` index for buffer chaining, and two flags: one to indicate whether the next field is valid and one controlling whether the buffer is read-only or write-only. This allows a buffer chain to contain both readable and writable parts (this is useful for implementing a block device).
The available ring consists of a free-running index (the *avail* index)[4], an interrupt suppression flag, and an array of indices into the descriptor array where each index references the head of a descriptor chain. As we can see, the available ring is separated from the descriptor array, adding another level of indirection: The avail index references an entry of the avail ring and this entry references an entry of the descriptor array (an head of a descriptor chain). This separation of the descriptor array from the available ring is due to the asynchronous nature of the virtqueue. In fact the hypervisor extracts the chains in the same order in which they have been inserted, but may process them in a different order and/or asynchronously. In this case some chains could require more time than others, and so the available ring could circle many times with fast-serviced chains while slow descriptors might still await completion (once again this is useful for block device implementation). The interrupt suppression flag is set when the virtio driver invokes the `disable_cb` and is suppressed when the `enable_cb` is invoked. In this way the guest can implement a mitigation strategy for hypervisor notifications (see section 4.2.1).
The used ring has the same structure of the the available ring, but is written by the host as descriptor chains are consumed. Moreover, the each entry of used ring contains not only an index in the descriptor array, but also a `len` field which is used

[3] e.g. In our work environment this are implemented as guest MMIO accesses.

[4] The ring index is a 16 bit unsigned integer, which is always incremented and is intended to overflow.

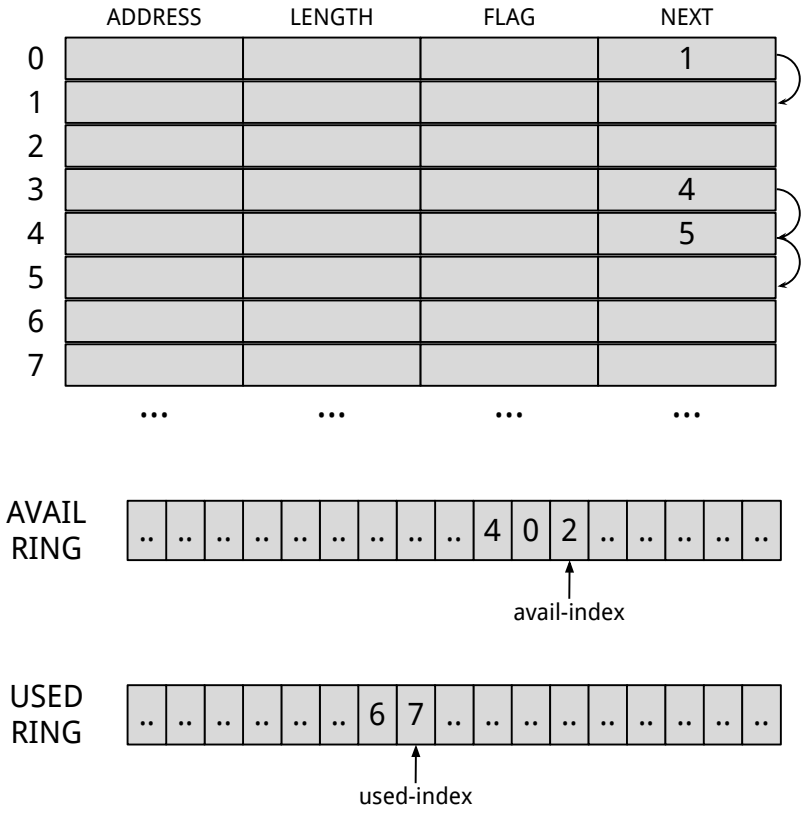

Figure 4.3: Virtio ring data structures, used to implement the Virtio interface. On the top the descriptor array is depicted as a table. On the bottom the *avail* ring and the *used* ring.

for input buffers: When an input buffer has been used, the hypervisor writes in this field the size of the read operation which has been done on the buffer itself.
The data structures are depicted in figure 4.3.

Putting all together, let's describe the journey of a virtio buffer in a virtual queue. A virtio driver invokes `add_buf` on the virtual queue, and the provided SG list is inserted in the descriptor array, using a free entry for each element in the SG list. All this descriptors are chained together. At this point a new entry is inserted in the avail ring, referencing the head of the chain just now built, and the avail index is incremented. Now the virtio driver can kick the hypervisor (if is the case), because there is a new entry in the avail ring. When the hypervisor sees that there is work to do, it pops the new entry from the available ring, processes it, pushes the used descriptor chain in the used ring, increments the used index and, if is the case, notifies the guest (e.g. send an interrupt). When the guest wants to get an used buffer, it invokes the `get_buf` method so that it can clean or complete to process

the returned buffer.

4.2.3 Minimizing notifications using Virtio

Let's see how we can build an Virtio driver and device emulator that try to minimize the notification rate in both directions, assuming we are dealing with our usual work environment (QEMUKVM as VMM and Linux as guest). As we've seen so far, in our work environment notifications are very expensive, and so minimizing them leads to big performance improvements. Since Virtio is explicitely designed to work in a virtualized environment, it's easy to write a driver that minimizes notifications. We will make an abstract example where the virtio driver, using a single virtual queue, receives requests from the kernel and passes these requests (represented as virtio buffers) to the hypervisor. The hypervisor processes these requests in a dedicated thread and returns the used buffers to the virtio driver. When the guest is notified by the hypervisor, the virtio driver gets the used buffers and do some post processing. To be more precise, we would like to minimize the ratio between the average notification rate and the average request rate, e.g. maximize the percentage of spared notifications.

The pseudocode for the virtio driver and the corresponding device emulation into the hypervisor is reported in figure 4.4. This pseudocode skips many details, but contains all the interesting strategies to minimize notifications. The request processing in the hypervisor is done in the IOThread, using a QEMUBH (section 2.2.1). The QEMUBH is scheduled in the `notification_callback` function, which is executed by a VCPU thread after a VMExit due to a guest notification. As we have already seen, the guest can notify the hypervisor using the `kick` method, which turns into a real notification (e.g. a VMExit) only if the hypervisor has the notifications enabled. The postprocessing is done by the NAPI polling function, which runs in a dedicated thread, and is scheduled by the virtqueue callback (`virtqueue_callback` function). The callback is executed in interrupt context when the hypervisor notifies the guest with an interrupt.

In conclusion, there is a dedicate worker both in the host (the QEMUBH) and in the guest (the NAPI polling function). Each worker runs in a separate thread, so that there is parallelism. When the guest notifies the host, the host worker starts its processing and stops only when there is no more work. When the host notifies the guest, the guest worker starts its processing and stops only when there is no more work. Note that the processing system is symmetric: There are two symmetric *producer/consumer couples* (in short PCCs). In the first PCC, the producer is the guest kernel which continuosly invokes `request()`, and the consumer is the QEMUBH, which continuously processes (comsumes) the requests. In the second PCC, the producer is the QEMUBH itself which invokes `virtqueue.push()` and the consumer is the NAPI polling function that does the post processing.

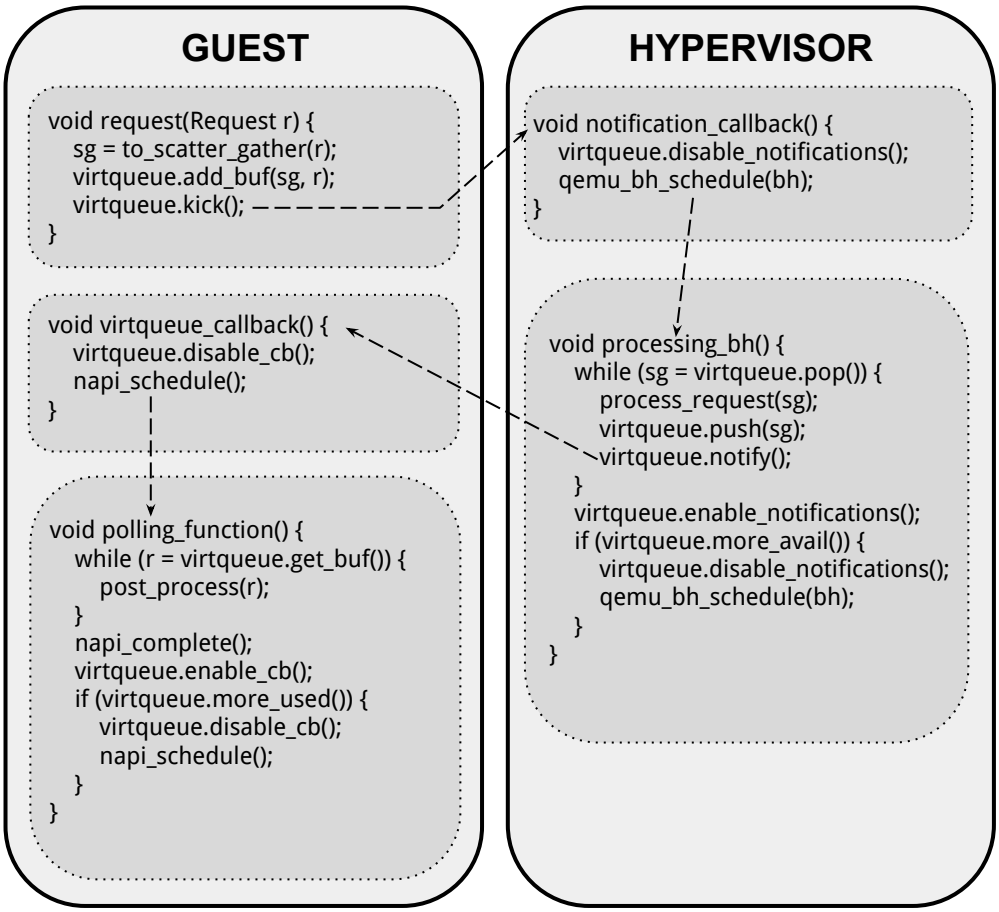

Figure 4.4: Pseudocode for the virtio driver and the virtio device emulation for the example in section 4.2.3.

How are the notification minimized? Using the very same ideas introduced by NAPI (see section 2.5.1): When a notification comes, if the notification rate is high because the incoming work request is high, we disable notifications and start polling for requests. While there are requests to process we keep processing, keeping the notifications disabled. When there are no requests left, we exit the working cycle and enable notifications, so that we can be notified in the future when new requests will come. This concept is employed for both the PCCs. To be precise, after the notifications have been enabled the consumer checks again if there is other work to do. If true the notifications are disabled and the consumer reschedules itself. This is done because of a race condition that we will show in the following "Race condition analysis" subsection. Using this system, if the consumer is slower than the producer (see the discussion repoted in section 3.1.2), the performance of the consumer/producer are very good.

Let's analyze how this system works, assuming the producer slower than the consumer in both PCCs. At the beginning notifications are enabled in both driver and device emulator. As the kernel invokes `request()` for the first time, the request is added to the virtualqueue and the kick notifies the hypervisor. Because of the notification, `notification_callback()` is invoked: Further guest notifications are disabled and the QEMUBH worker is scheduled. The QEMUBH worker starts processing requests and keep doing it as the virtqueue avail ring is not empty (and because of our assumptions this is very unlikely to be empty). After the first request has been processed, the host successfully notifies the guest (`virtqueue.notify()`), and therefore `virtqueue_callback()` is invoked: Further host notifications are disabled and the NAPI polling function is scheduled. The polling worker starts to post-process requests and keeps doing it as the used ring is not empty (and it is unlikely to be so because of our assumptions).

As we can see, this system can, in theory, work without notifications, except for the first two ones. Of course this is not realistic, also because a consumer could be faster than a producer. For instance, the guest consumer (NAPI polling function) is often a fast one, and so the NAPI could not moderate adequately the host notifications. In this situation, some other form of moderation is necessary to get good performance.
Moreover, the `request()` function could implement some form of stream control and tell the kernel to stop sending requests when it sees that the avail ring is full. If requests stop, also the two consumers stop and notification must be enabled, otherwise we couldn't tell the kernel to restart requesting. When the guest is notified again, the kernel can restart sending request (causing another notification). Nevertheless, the notification rate can still be very low w.r.t. the request rate.

Race condition analysis

The consumer working cycle exits when there is no work left. Once exited, the notifications are enabled. The race exists because these two operations, namely checking for more work and enabling notifications are not atomic. If in beetween these operation the producer inserts more request in the virtual queue and tries to notify the consumer, the notification of this last request is not done because notifications are disabled. If the consumer does not double check for more work after enabling notifications, and the producer doesn't add other requests for an hour, the last request stalls in the queue for an hour. Since the consumer double checks, however, it sees that new requests have come while it was not polling and reschedules itself (and possibly consume the request immediately). In this way requests cannot stall. Figure 4.5 shows an example of race.

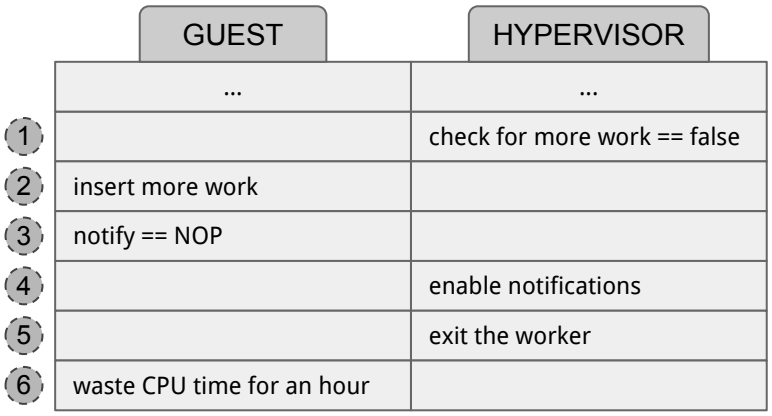

Figure 4.5: Race condition between a producer and a consumer. The race condition exists because the consumer doesn't have a way to check for more work and enable notifications in an atomical manner.

Other details

In our example we've skipped many details in order to keep the pseudocode simple. However, it's important to note that both the NAPI polling function and the QEMUBH, in their working cycle, must limit the amount of work done, otherwise they would monopolize the VCPU thread (in the NAPI case) or the IOThread (in the QEMUBH case). For these reason the workers must count the requests processed and exit the working cycle when the count exceeds a given amount. In the NAPI case, this amount is passed to the polling function in the `budget` argument, whereas in the QEMUBH case we have to choose a value (e.g. 256 is a good compromise between performance and responsiveness of the event-loop).

4.3 An efficient Virtio network driver

Using the idea presented in section 4.2.3 one can build a very efficient virtio network driver and network device emulator (*virtio-net*). The driver source can be found in the linux kernel sources (drivers/net/virtio_net.c), while the device emulation can be found in the the QEMU source (hw/virtio-net.c). Since a network adapter deals with two independent data streams, one for TX and one for RX, two virtual queues are employed. The virtio block device uses just one virtual queue because the two streams are not independent, but are always request and response.

For each virtual queue, two PCCs can be defined: the *direct* PCC and the *inverse* PCC, depending on who is the communication master. On the TX path the guest is the master: in the TX direct PCC the guest produces TX avail buffers to send

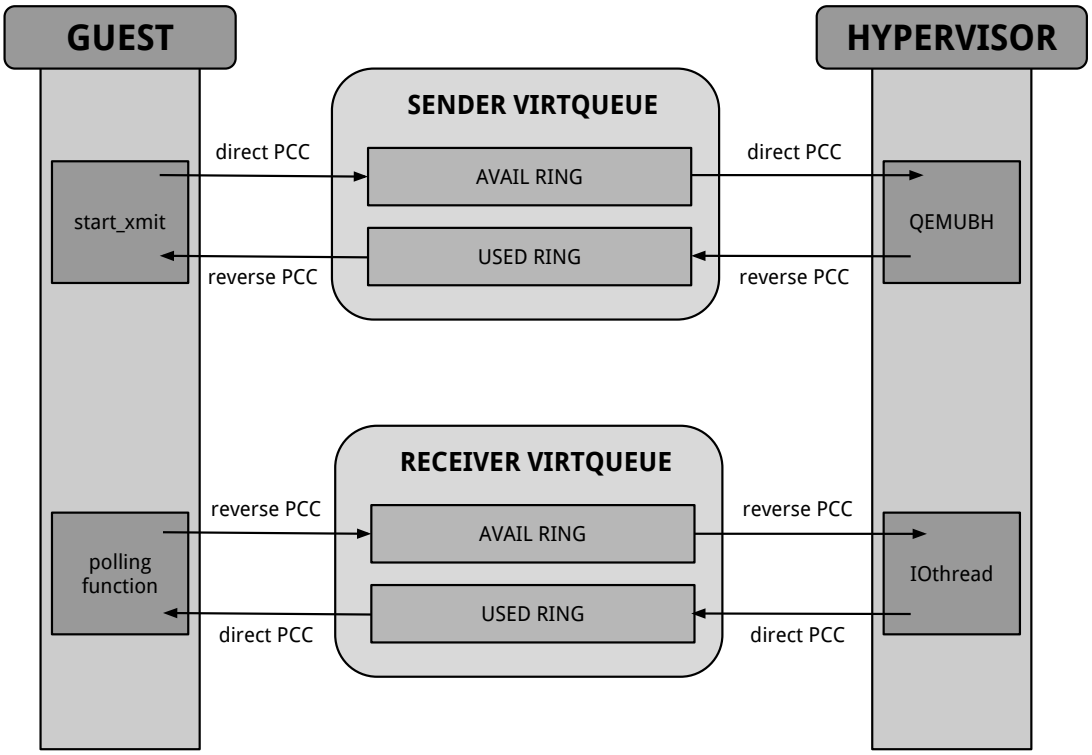

Figure 4.6: High level scheme of the virtio-net implementation. On the top there is the sender virtual queue with its two Producer/Consumer Couples (PCCs), which handle the TX path. On the bottom there is the receiver virtual queue with its PCCs, which handle the RX path.

and the hypervisor consumes (sends) avail buffers, while in the TX reverse PCC the hypervisor produces used buffers and the guest consumes (frees) used buffers. On the RX path the hypervisor is the master: in the RX direct PCC the hypervisor produces (receives into) RX used buffers and the guest consumes (receives from) RX used buffers, while in the RX reverse PCC the guest produces RX avail buffers and the hypervisor uses RX avail buffers. This situation is depicted in figure 4.6.

4.3.1 TX path

The TX path uses the sender virtual queue (`svq`), where the virtual queue callback is initially disabled on the driver side. When the kernel invokes the `ndo_start_xmit` method, it does the following

1. Free any pending used buffers in the used ring repeatedly calling `svq.get_buf()`. The token returned is a pointer to a `sk_buff` that was previously added to the avail ring, so that the `sk_buff` can be freed with `dev_free_skb(skb)`.

2. Convert the sk_buff provided as argument to a SG array. The SG array will have more elements if the sk_buff is nonlinear or just an element if linear.

3. Invoke svq.add_buf(sg, skb) to insert a new output SG in the avail ring.

4. Invoke svq.kick() to notify the hypervisor.

5. Call skb_orphan(skb) so that the kernel won't wait for the sk_buff to be freed and so won't invoke the TX watchdog.

6. If there is no space in the virtual queue for the next send, tell the kernel to stop making new TX requests by calling netif_stop_queue() and invoke svq.enable_cb_delayed()[5]. We will be notified by the hypervisor when this pushes more used buffer to the used ring, so that we can call netif_wake_queue() and the kernel can make new TX requests. After notification are enabled we must do a double check for more used buffers otherwise we would run into the same race condition described in section 4.2.3. If there are more used buffers, we free them (in the same way as we do at point 1) and, if now there is space enough for a new send, we disable the callback and tell the kernel it can restart making TX request (invoking netif_wake_queue()).

Note that while the avail ring is not full (e.g. while the add_buf is successful) the sender virtual queue callback is never used, since the used buffers are cleaned at the beginning of the the ndo_start_xmit method. However, when there is no more space, the kernel is told not to call this method anymore, and so the only way to free used buffer after that is to use the callback.

The device emulator part of the TX path is implemented with the same scheme presented in section 4.2.3 (see figure 4.4), where the guest notification are initially enabled. In this way the guest notification are moderated when the guest sender is faster than the QEMUBH TX processing cycle. The TX processing is done basically calling qemu_sendv_packet(sg)[6], where sg is the SG list extracted by the avail ring. A copy is not performed in the hypervisor, but is necessary to map the guest physical memory onto the hypervisor virtual memory, since the virtio descriptors contains physical addresses. This mapping, however, is done internally by the virtqueue.pop(), so it's not visible to the device emulator.

4.3.2 RX path

The RX path uses the receiver virtual queue (rvq), where the virtual queue callback is initially enabled on the driver side. At initialization time, the driver must provide the hypervisor with some receive buffers, otherwise it cannot accept incoming

[5]This is a variant of the enable_cb method that delays the abilitation in the future. The variant is implented only if the Virtio EVENT_IDX feature is set. However, you cannot specify how far in the future the abilitation will take place.

[6]This is a simple variant of the qemu_send_packet() function, that accepts a scatter-gather array.

packets[7]. In other words, the driver has to fill the avail ring with receive buffers calling `add_buf()` as many times as possible. A pointer to an empty `sk_buff` is passed as token.

On the device emulation side (virtio-net frontend), the guest notification are initially enabled. When the hypervisor receives a packet, the network backend invokes the `virtio_net_receive` method, that performs the following actions:

1. If there are no receive buffers to use, enable guest notifications, so that we will be notified when more receive buffers are added to the avail ring. Then return 0 (the number of bytes received). When the QEMU network core sees that the virtio-net frontend is not able to receive the packet, it puts the received packets into an internal queue (see section 2.2.4). When the guests notification comes, the frontend invokes `qemu_flush_queued_packets()`, so that the QEMU network core tries again to invoke `virtio_net_receive` on each queued packet. After we enable guest notifications, we must double check for more avail buffers, otherwise the usual race condition can show up.

2. Pop a receive buffer from the virtual queue (`rvq.pop(sg)`).

3. Copy to this receive buffer the ethernet frame passed to `virtio_net_receive`.

4. Pushes the used buffer to the used ring. To be precise, the push operation is not performed with the `push` method, but in two (or more) splitted calls. Firstly the `fill` method is called to put an used SG list in the ring without updating the user ring index, and then the `flush` method is called to update the used ring index. The reason for this is that in general we need more avail SG lists (and so more `pop`) for a single ethernet frame. Since we don't want to expose an used index that corrispond to a incomplete ethernet frame we can update the used index only when the received packets is completely in the used ring.

5. Notify the guest with `rvq.notify()`.

The driver part of the RX path is implemented with the same scheme presented in section 4.2.3. The NAPI polling function is scheduled by the receiver virtual queue callback, which also disable further callbacks. The polling function polls for used receive buffers calling `rvq.get_buf()`, that returns the `sk_buff` token. Each used buffer is processed by the `receive_buf()` function, which invokes the `netif_receive_skb()` as last operation. When the working cycle exits, it means that the NAPI budget is over or that no more used buffer are ready. At this point the polling function has made room in the avail ring and so refills it with new receive buffers[8].

[7] We've seen the same problem with e1000 in section 2.5.4.

[8] Observe that the receive buffer management is very similar to the one employed by e1000.

After that the callbacks are enabled and the usual double check performed (with callbacks disabling and NAPI rescheduling if there are more used buffers).

4.3.3 Other details

In this section we've outlined the most important details about *virtio-net* implementation. However, we have skipped some details such as out-of-memory (OOM) managment, and receive buffer allocation/deallocation and processing. In fact three types of receive buffers can be used: *small* buffers, *big* buffers and *mergeable* buffers. For further information refer to the source code.

Another important feature is the use of TSO/GSO[9] kernel support (introduced in section 2.3), that allows to transmit and receive very big packets (up to about 64KB). These features lead to huge improvements in TCP throughput with reference to a case in which TSO/GSO packets are not used.

4.3.4 Performance analysis

In this section we will preform the same experiments presented in section 3.1, so that we can compare the result with the e1000 performance (patched or unpatched).

TX performance

The results in 1-VCPU case are shown in table 4.1. We can see that the interrupt rate is very low and the performance is good ($\sim$ 160 Kpps). At the same time, the TX notification rate is almost zero.
What happens here is that the producer (the guest) is faster than the consumer (the hypervisor) in the TX direct PCC: This is why the hypervisor (almost) never needs to exit from its working cycle and so (almost) never enables guest notifications.

Since the guest is faster, the sender avail ring should be always close to full: As soon as the hypervisor consumes some avail buffers, the guest should replenish the avail ring, tell the kernel to stop sending packets, and enable interrupts. In the TX reverse PCC, in other words, the consumer (the guest) is faster than the producer (the hypervisor). In this scenario, therfore, the interrupt rate should be very high, because the system would enter in a permanent *almost-full* state:

1. The sender queue avail ring has space for only one (or a very few) packet, therefore rapidly replenishes the ring and enables the interrupts.

2. As soon as the hypervisor processes the next TX packet (making room in the queue), it notifies the guest. The cycle go on from point 1.

[9] Generic Segmentation Offload is a network driver feature that extends the TSO concept to other protocols, such as UDP.

Virtio-net	**1-VCPU**	**2-VCPUs**	
Interrupt rate	0.822	0.8	KHz
TX packet rate	158.1	154.8	Kpps
TX bitrate	130.3	127.6	Mbps
TX notifications	0.007	0.007	Mbps

Paravirtualized e1000	**1-VCPU**	**2-VCPUs**	
Interrupt rate	0.37	0.58	KHz
TX packet rate	183.4	182.9	Kpps
TX bitrate	135.1	133.1	Mbps
TX notifications rate	0.4	0.34	KHz
MMIO write rate	1.2	1.4	KHz
MMIO read rate	0.4	0.55	KHz

Table 4.1: Guest to host statistics with paravirtualized solutions. The table on the top shows virtio-net performance, while the table on the bottom shows paravirtualized e1000 performance. Each table reports a set of measurements for the 1-VCPU case and one for the 2-VCPUs case.

In this situation we should have nearly one interrupt per packet, and so bad performance. However, from table 4.1 we don't see this circumstance because of the way the interrupts are enabled. As we have seen in section 4.3.1, the `enable_cb_delayed` method is used in place of `enable_cb`. In this way the interrupts are enabled after a while, and not immediately. In the current implementation, the interrupt are enabled when the hypervisor has processed $\frac{3}{4}$ of the pending packets in the queue[10]. Therefore when an interrupt comes, 75% of the queue is empty, and not just one packet. In other words, the system enters a state in which the guest is notified every 192 packets[11], and still the queue is never empty (so the hypervisor never stops) because the guest is fast to replenish the queue. In conclusion we should expect an interrupt rate of $\frac{158.115Kpps}{192} = 0.823$ KHz, which is exactly what we have in the table.

Although not implemented with timers, the `enable_cb_delayed` method is just a different *incomplete* form of interrupt moderation. It is used to moderate the notification rate within a PCC where the consumer is faster than the producer: In our case the producer is the hypervisor, which produces used buffers, and the consumer is the driver, which consumes (free) used buffers. In order to see the mitigation effects, we have tried to run the same experiment with a modified virtio-net driver, where `enable_cb` is used in place of `enable_cb_delayed`. The results

[10] In our case the queue is full of pending packets.

[11] The current default size for a virtual queue is 256, and so $256 \cdot 0.75 = 192$.

Interrupt rate	108.4	KHz
TX packet rate	110.4	Kpps
TX bitrate	91.0	Mbps
TX notifications	0.007	Mbps

Table 4.2: Guest to host statistics with 1 VCPU per guest, when the `enable_cb_delayed` method is not implemented

are shown in table 4.2.
As we can see, here the situation is exactly the one we described previously (the almost-full state), with an interrupt for each TX packet. The TX packet rate oscillates between 80 Kpps and 150 Kpps, the system is very unstable and has a very low responsiveness: Interrupt mitigation was definitely required in this case.

It's very important to observe that you cannot always use this form of mitigation, because of its incompleteness: there is not a mechanism to force timely notification of pending events. Even worse, it can be the case that a pending event stalls forever in a queue. In our case, the TX reverse PCC, there is not such a problem, for two reasons:

- On the TX path, the comunication *master* is the guest, and so when it calls `enable_cb_delayed()` we are sure that the pending TX buffers will be processed by the *slave* (the hypervisor) as soon as possible, and so the interrupt will be sent for sure, e.g. the pending events cannot stall.
- Considering the way the TX path is implemented (see section 4.3.1), we don't care about having used TX buffers freed as soon as possible, since they are basically freed when needed.

In the RX path we are not so lucky (see the next subsection).

The results for the 2-VCPU test case are shown in table 4.1, and are very similar to results for 1-VCPU case, since there is no significant work that can be done in parallel.

RX performance

The measured critical rate is about 110 Kpps, which is not very high if compared with what we get with e1000 (adding the mitigation patch). The collected statistics are shown in table 4.3.
Also in this case, the reason for low performance is tied to the lack of interrupt mitigation. As we can see, the interrupt rate is very high (57 KHz), because Virtio does not limit the interrupt rate in any way. From an high interrupt rate we can infer that in the RX direct PCC, the consumer (the NAPI) is faster than the producer

Virtio-net	**1-VCPU**	**2-VCPUs**	
Interrupt rate	57.6	43.5	KHz
RX packet rate	103.0	100.1	Kpps
RX stream	127.6	124.0	Mbps
RX notification	0.007	0.004	Mbps

Paravirtualized e1000	**1-VCPU**	**2-VCPUs**	
Interrupt rate	3.8	3.7	KHz
RX packet rate	317.3	315.4	Kpps
RX bitrate	231.0	229.6	Mbps
RX notifications	0.001	0.002	Mbps
MMIO write rate	7.5	7.4	KHz
MMIO read rate	3.8	3.7	KHz

Table 4.3: Host to guest statistics with paravirtualized solutions. The table on the top shows virtio-net performance, while the table on the bottom shows paravirtualized e1000 performance. Each table reports a set of measurements for the 1-VCPU case and one for the 2-VCPUs case.

(hypervisor): This results in only about 2 packets processed by the polling function for each interrupt. We could add mitigation in the same way we have added it in the TX inverse PCC (see the previous subsection): Using the `enable_cb_delayed` method in place of the `enable_cb` method when exiting from the NAPI working cycle. However, this simply cannot be done:

- On the RX path, the master is the hypervisor and so the guest (the slave), when calling `enable_cb_delayed()`, cannot know when the next packet will be received by the hypervisor, and therefore cannot know when a total of 192 packets will be received. Since the Virtio mitigation is incomplete, up to 191 RX packets could stall in the used ring. Of course, this is not acceptable.
- Latency is affected negatively by the mitigation in this case.

A complete form of mitigation is here the only solution, but this has not been implemented yet.

On the RX reverse PCC the consumer (hypervisor) is slower than the producer (guest) and consequently the RX notification rate is very good (about zero notifications).

The measured results for the 2-VCPU case are shown in table 4.3. There is not

a substantial difference with the 1-VCPU case, because there is not enough parallelism in the guest to exploit.

4.4 Porting paravirtualization to e1000

In the previous sections we've seen how is possible to get good performance using a paravirtualized I/O solution. Virtio drivers are efficient because the only make use MMIO accesses and interrupts to notify the other peer of a virtual queue, and exchange data and metadata through shared memory. However, the Virtio interface must also be general enough to support different kind of devices and is not specifically tailored to the network device world: Most of the complications come from the need of a paravirtualized disk device. As an example, a network driver doesn't need two rings (avail and used) for each virtual queue, and doesn't even need to separate the ring from the descriptor table. The descriptor table and two indices in that table would have been enough. Moreover, a complete mitigation scheme is not implemented, and this is a limitation in same situations (see section 4.3.4, RX performance), Lastly, the Virtio solution requires a completely new set of device drivers/emulators, which is a greater effort than modifying an existing device driver/emulator.

In this section we will see how is possible to extend the e1000 interface in order to adopt the paravirtualization paradigm. As usual, we will try to minimize the modifications to the existing code, so that the proposed patch can be easy to test and mantain.

4.4.1 Asynchronous TX Processing

In the e1000 emulation we've seen so far, the TX processing is done by a VCPU thread (section 2.4.1). A better solution however, is to employ the processing/notification scheme presented in section 4.2.3 (e.g. NAPI-like moderation), in which the hypervisor processing is done asynchronously using a QEMUBH. In this way the TX processing is executed by the IOThread in parallel to the VCPUs.

4.4.2 Communication Status Block

In order to implement an efficient communication scheme we have to remove, where possible, MMIO accesses that exchange data, leaving only accesses that implement guest notifications to the hypervisor. As an example, writes to TDT and RDT registers are used for two functions:

1. (notify) Notify the hypervisor that there is something new to process (new Tx descriptors to process or new Rx descriptors to use).

2. (status) Tell the hypervisor which descriptors have just been added.

A notification (1), however, is not always necessary[12], while status information exchange (2) is always desirable, so that the other peer is always updated and it's less likely to exit from its working cycle. It would then be better to split these functions: TDT/RDT writes should be used only for notifications, while status exchange can be done using shared memory. For these reasons a dedicate block of shared memory, called *Communication Status Block* (CSB), has been allocated in the guest memory. The CSB is a set of 32-bits words that are used to exchange information about the status of the communication without VMExits.

The CSB contains the following words:

- TXSNTS (Tx Software Next To Send): is synchronized with the `tx_next_to_use` variable (when in a coherent state). In short, where the `ndo_start_xmit` starts to insert new TX descriptors in the TX ring. It is used to replace the status information function of the TDT register.
- TXHNTS (Tx Hardware Next To Send): is synchronized with the TDH register. In short, where the hardware takes the next TX descriptor to process.
- TXSNTC (Tx Software Next To Clean): is synchronized with the `tx_next_to_clean` variable. In short, where the TX interrupt routine takes the next used TX descriptor to clean.
- RXSNTP (Rx Software Next To Prepare): is synchronized with the `rx_next_to_use` variable. In short, where the driver adds new RX descriptors to use for reiceving new frames.
- RXHNTR (Rx Hardware Next To Receive): is synchronized with the RDH register. In short, where the hardware takes the next RX descriptor to use for an incoming frame.
- RXSNTR (Rx Software Next To Receive): is synchronized with the `rx_next_to_clean` variable. In short, where the RX interrupt routine takes the next used RX descriptor to push the received frame to the network stack.
- TXNTFY (Tx Notify Flag): Set/Cleared by the hardware to enable/disable TX notifications. At the end of the `ndo_start_xmit` method, the driver always updates TXSNTS, and updates also TDT only if TXNTFY is set.
- RXNTFY (Rx Notify Flag): Set/Cleared by the hardware to enable/disable RX notifications. After adding new RX descriptors in the ring, the driver always updates RXSNTP, and updates also RDT only if RXNTFY is set.

Some words (TXSNTS, TXSNTC, RXSNTP and RXSNTR) are always written by the software (the driver) and read by the hardware (the device emulator). The other

[12]For example, it's not necessary if the other peer is in its working cycle.

words (TXHNTS, RXHNTR, TXNTFY and RXNTFY) are always written by the hardware and read by the software.

How we can see, these variables capture in each moment the status of the communication (both RX and TX). Most of them are just shadow copies of registers (TXHNTS and RXHNTR) or shadow copies of existing driver variables (TXSNTS, TXSNTC, RXSNTP and RXSNTR), therefore nothing new so far. Shadow copies of registers are useful because the driver can read them without a VMExit. Shadow copies of existing variables are useful only for technical reasons: they allow us to have all the interesting variables in a contiguous chunk of physical memory so that the driver can to tell the hardware only the CSB physical address. Otherwise the driver should tell the hardware a different physical address for each interesting variable[13].

On the other end, TXNTFY and RXNTFY are something new. These are used by the hardware to disable TDT writes and RDT writes when these are not necessary. As we have outlined in section 4.4.1, the idea is to implement the usual producer/consumer scheme (NAPI-like moderation), which requires a method for the consumer to disable producer notifications.

Similarly to what happens to the other shared memory structures (e.g. TX/RX rings), the hardware must be told where the CSB has been allocated in the guest physical memory. For these reason, two 32-bit registers, CSBBAH (CSB Base Address High) and CSBBAL (CSB Base Address Low) have been added to the register set. The driver must write the physical address of the CSB to these registers.

4.4.3 Implementation

This section briefly reports what changes have been added to the RX and the TX path to implement e1000 paravirtualization. The changes are relative to the existing implementation of e1000, possibly improved with the moderation patch. As we have already observed, interrupt moderation is very useful within PCCs in which the hypervisor is the producer, and the guest is is faster than the hypervisor. However, the paravirtualization patch is independent on the moderation patch. The driver batching mechanism, on the other hand, is not intended to be used jointly with paravirtualization, simply because this would not make any sense: Since paravirtualization exports the status of the communication, there's no need to flush pending TX descriptors, we just have to notify the hypervisor when necessary.

A new parameter, `paravirtual`, has been added to the e1000 module. The parameter value can be specified only at module loading time and cannot be changed

[13] In the existing e1000 driver the memory for these variables is not allocated all together, so the driver data memory is not necessarily contiguous.

dynamically. When `paravirtual` is 0, the driver is the original one, and the device emulator (e1000 frontend) behaves as usual[14]. When `paravirtual` is 1, the driver allocates the CSB, initializes it, and writes its physical address to the CSBBAH and CSBBAL registers. On the hypervisor side, the emulated e1000 hardware switches to "paravirtual mode" when the driver writes to the CSBBAL register and CSBBAL/CSBBAH registers specify together a non-null address.

At initialization time, the CSB words are set to 0, except for TXNTFY and RXNTFY that are initialized to 1.

Changes in the TX path

At the end of the `ndo_start_xmit` method in the e1000 original driver, where the TDT would be updated, we changed the code so that TXSNTS is updated, while the TDT is updated only if TXNTFY is set. In the TDT write handler, TXNTFY is cleared to disable further notifications and the QEMUBH is scheduled. The QEMUBH handler invokes the `start_xmit` function, so that the TX processing (working) cycle is entered. The working cycle can exit because we processed too many descriptors (256 in the current implementation) or because there is no more work to do. If the working cycle has processed at least one descriptor, the QEMUBH is rescheduled. Otherwise TXNTFY is set to 1, and the usual double check is performed. If new work is found, TXNTFY is cleared and the QEMUBH rescheduled. If new work hasn't come yet we don't reschedule the QEMUBH. Note that we reschedule even if just one descriptor has been processed, and not only if a full burst (256) of descriptors has been processed: This is a form of *spinning* that can be useful when dealing with fast backends, because we make the QEMUBH more likely to be rescheduled, and so the TX notification more likely to be disabled. The spinning can work well in this case because, after rescheduling, the QEMUBH handler is not executed immediately, but in the next event-loop iteration, so that the guest has some time to post more work in the queue.

Changes in the RX path

At the end of `e1000_alloc_rx_buffers` in the original driver, where the RDT would be updated, we changed the code so that RXSNTP is updated, but RDT is updated only if RXNTFY is set. In the RDT write handler, if the RDT write actually gives some new receive buffers, RXNTFY is cleared to disable further RX notifications. Remember that in this last case `qemu_flush_queued_packets()` is called (see section 2.2.4). When the hardware runs out of receive buffers, the e1000 `can_receive` method returns 0. Before returning, however, it has to set RXNTFY, so that when new receive buffers are put in the RX ring, a RDT write can flush the queued packets (if any). After notifications have been enabled, as usual, a double check is performed in order to avoid race conditions.

[14] To keep the code simple, the CSB is always allocated even if `paravirtual` is 0.

Other changes

Since we know to be in a virtual machine environment, some more optimizations can be done. In particular, after writing to IMS and IMC registers (see section 3.1.1), we can avoid reading the STATUS register to flush the previous register write, since there's no need to do so on the emulated hardware.

4.4.4 Improvement analysis

In this section we will perform the usual experiments (see section 3.1), so that we can evaluate the performance improvements. The e1000 module has been loaded with the following parameters:

Parameter	Value
TxIntDelay	0
TxAbsIntDelay	0
InterruptThrottleRate	4000
batching	0
paravirtual	1

TX performance

The results for 1-VCPU case are shown in table 4.1.
As we can see, performance are better with reference to the previous tests (both e1000 and Virtio). The interrupt rate is very low, thanks to mitigation and NAPI. The TX notification are minimized because of the usual NAPI-like scheme. Note that the total MMIO access rate is 1.611, which is about 4.35 times the interrupt rate: 3 MMIO accesses per interrupt are due to the ICR read and IMC and IMS write, while 1.35 access per iterrupt are due to the TX notifications.

The results for the 2-VCPU test case (shown in table 4.1) are very similar.

RX performance

In the 1-VCPU case the measured critical rate is about 315 Kpps, which is the best result so far. Table 4.3 shows the statistics when the VM accepts an incoming rate of about 317 Kpps.
The very good result is due to the interrupt moderation, the NAPI, the MMIO access optimizations (avoid reading the STATUS register to flush the register write) and the RX notification moderation. The total MMIO access rate is 11.318 KHz, which means about $\frac{11.318KHz}{3.774KHz} \sim 3$ MMIO access per interrupt. These three interrupt are the ICR read, and the IMC and IMS writes. RX notifications are essentially absent, because these notifications are enabled only when the RX ring has not receive buffers to use, e.g. when the hypervisor start to be faster than the guest on

the RX path. In our case the incoming rate is still too low to make that situation very likely.

With 2 VPCUs the measured critical rate is also about 315 Kpps. Table 4.3 shows the statistics when the VM accepts an incoming rate of about 315 Kpps.

Chapter 5

Conclusions and future work

In this work we have shown how is possible to boost the packet-rate performance of an e1000 network adapter in a Virtual Machine environment. First of all we analyzed the existing implementation and we pointed out its problems and bottlenecks. Then we proposed a patch to the QEMU e1000 frontend (the *moderation* patch) and a patch to the Linux e1000 driver (the *batching* patch) that can be used, independently on each other, to improve performance. Both patches are very simple and don't extend the e1000 interface specification. After that we analyzed the Virtio I/O paravirtualization framework and provided a small extension to the e1000 interface that allows to remove some inefficiency by porting the I/O paravirtualization concepts to the e1000 platform.

Comparing the TX and RX performance of the existing e1000 implementation with our best results (e1000 paravirtualization) we have obtained a 8.9× speed-up on the TX path and a 22× speed-up on the RX side. This packet rate performance is comparable or superior to the Virtio performance and is obtained with small modification to the e1000 driver and device emulator.
A short paper ([12]) collecting the ideas presented in this work, in combination with the VALE ([11]) fast software switch, has been submitted to USENIX ATC'13.

More work can be done to achieve further improvements with the e1000 platform. In particular, three aspects can be optimized:

- Packet-rate: Although this parameter is the optimization objective of these thesis, we can further improve performance removing some packet copies. For example, the current e1000 device emulation copies a TX packet from the guest memory to a local buffer: this copy is essentially useless and it can be avoided with some memory mapping efforts.

- Latency: Interrupt moderation worsens latency, so we should implement some euristics aimed at bypassing moderation delay when the network traffic requires low latency more than high packet rate (e.g. HTTP transactions).

Moreover, MSI/MSI-X ([8]) interrupts can be used in place of the classic interrupt mechanism.

- TXP throughput: The key for performance here is big packets, more than high packet rate. The bigger packets you can make, the higher throughput you can get. TSO/GSO features must be exploited, like Virtio does.

Bibliography

[1] The QEMU project. http://www.qemu.org.

[2] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization.

[3] Agesen, O., Mattson, J., Rugina, R., and Sheldon, J. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 35–35.

[4] AMD. *Secure Virtual Machine Architecture Reference Manual*, May 2008.

[5] Bellard, F. Qemu, a fast and portable dynamic translator.

[6] Chisnall, D. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[7] Corporation, I. *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developers Manual*.

[8] Emulex. Msi and msi-x: New interrupt handling improves system performance with emulex lpe12000 8gb/s pcie fibre channel hbas.

[9] Huang, Q. An introduction to virtual machines implementation and applications. *SIGOPS Oper. Syst. Rev. 42*, 5 (2006), 5–14.

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

[11] Rizzo, L., and Lettieri, G. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 61–72.

[12] Rizzo, L., Lettieri, G., and Maffione, V. Revisiting virtualized network adapters.

[13] Russell, R. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 95–103.

[14] Smith, J. E., and Nair, R. *Virtual Machines - Versatile Platforms for systems and processes*. Elsevier, San Francisco, CA, USA, 2005.

[15] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 1–14.

[16] VMWare. Performance evaluation of vmxnet3.

[17] Wang, J., Wright, K.-L., and Gopalan, K. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing* (New York, NY, USA, 2008), HPDC '08, ACM, pp. 109–118.

[18] Yehuda, B. Utilizing iommus for virtualization in linux and xen.