Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Systems Lab Report

Systems Group, Department of Computer Science, ETH Zurich

Using virtualization for PCI device drivers

by

Reto Lindegger, Lukas Humbel, Daniela Meier

Supervised by

Simon Peter

September 19, 2012

inf | Informatik
Computer Science

Abstract

A major obstacle when introducing a new operating system is device support. We analyze the possibility of using a virtualized Linux to provide driver support for PCI devices, by extending the Barrelfish virtual machine VMKit. As it turns out to be impossible to create a generic PCI passthrough without any hardware support (IOMMU), we focus our work on one specific network card. For allowing DMA to work correctly, we translate the physical addresses in software. To access the device from Barrelfish we provide a virtual network adapter and use the Linux Ethernet bridging to pass packets. Our measurements show an order of magnitude smaller throughput than a native driver solution, but still there's room for improvement left.

Contents

1 Introduction **3**

2 Design **3**
2.1 Overview . . . 3
2.2 Hardware access for the device driver . . . 4
2.3 Communication with Linux . . . 4
2.4 DMA Implementation Alternatives . . . 4
2.4.1 Paravirtualization . . . 5
2.4.2 Translation . . . 5
2.4.3 IOMMU . . . 5
2.4.4 Discussion . . . 5

3 Related Work **6**
3.1 PCI Passthrough . . . 6
3.2 Guest - Host Communication . . . 6

4 Implementation **6**
4.1 Overview . . . 6
4.2 Map real PCI hardware into the VM . . . 6
4.2.1 PCI Subsystem . . . 7
4.2.2 PCI Configuration Space . . . 7
4.2.3 Device registers and interrupts . . . 8
4.2.4 Handling DMA . . . 8
4.3 Guest - Host communication . . . 9
4.3.1 Setup . . . 9
4.3.2 Transmit and Receive Descriptor Tables . . . 11
4.3.3 Packet transfer from Linux to Barrelfish . . . 11
4.3.4 Packet transfer from Barrelfish to Linux . . . 11
4.4 Bridging . . . 12

5 Performance Evaluation **12**
5.1 Throughput . . . 12
5.2 Latency . . . 12
5.3 Discussion . . . 13

6 Known Issues **15**

7 Summary and Further Work **15**

A System Setup **17**

B PCI configuration space **17**

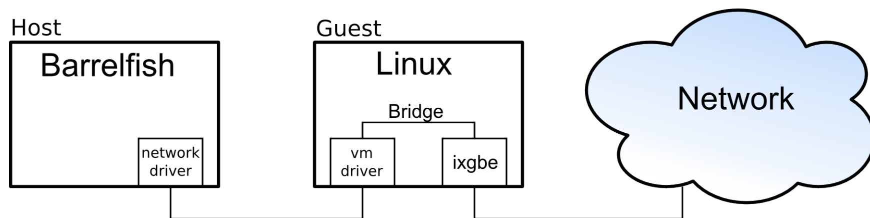

Figure 1: Setup and message flow overview

1 Introduction

Barrelfish is a research operating system mainly developed at ETH Zurich. This circumstance allows great freedom in design and implementation. However, this also has a disadvantage. Since everything has to be written from scratch, a new device driver has to be written for every piece of hardware one wants to use. On the other hand, Linux has a large collection of drivers and a global developer community constantly writing and improving device drivers. It would be really convenient, if these Linux drivers could be used with Barrelfish. The idea is to use a virtual machine to run Linux and use it's device drivers. The good news is that there is already a virtual machine available. For his master thesis, Raffaele Sandrini wrote a virtual machine for Barrelfish called VMKit [1] and already got Linux running in it. Nevertheless, it does not include a PCI subsystem nor direct access to any hardware. It provides only the minimal functionality necessary to allow Linux to run. What we need is a way to grant Linux access to PCI devices. To do this, we need to emulate a PCI system and provide a pass-through PCI device over which Linux communicates with the real hardware. But allowing the virtualized Linux to access the physical hardware doesn't automatically allow the Barrelfish host to access the device. There has to be a communication channel from Barrelfish into the virtual system.

2 Design

2.1 Overview

There are several necessary steps to use a standard Linux device driver with Barrelfish. The design can basically be split into two different parts. On one side, there is the Linux driver (ixgbe) which interacts with the real hardware. Furthermore, Barrelfish needs to be able to communicate with this driver to use the functionality of the hardware device. Therefore a communication channel from Barrelfish into the virtual machine is needed: The vm driver. This design layout is shown in Figure 1.

2.2 Hardware access for the device driver

As already mentioned, the Linux driver needs access to the real hardware. Since we are talking about PCI devices, this means the Linux driver must be able to access the PCI configuration space of the said device, access device registers which are usually memory mapped and receive the correct interrupts. First, the virtual machine monitor has to get control over the PCI device. To achieve this, it registers a driver for the device in Barrelfish. As a next step, the VM has to emulate a PCI subsystem and a virtual PCI device which imitates the desired hardware. The Linux inside the VM sees the virtual PCI device and can load a module for it. Read and write access from the Linux driver to the virtual device can now be intersected and redirected to the real hardware.

2.3 Communication with Linux

A working Linux driver inside a virtual machine which can access the real hardware is nice, but it isn't of much use for the host system. Barrelfish has to be able to control the Linux driver from outside the virtual machine, allowing it to use the functionality of the hardware without really implementing a specific device driver. In the case of a network card, the desired functionality would be that we can send packets from Barrelfish (host system) to the Linux (guest system) which then sends them over the real network card to the connected network. Of course the inverse direction should also be supported. To achieve a connection between Barrelfish and Linux, we introduce a new virtual network device for the VM and a Linux driver for this virtual device. Inside the VM, the driver can register itself as a network driver, therefore this virtual device and the real network device can be bridged using the Linux bridging functionality. The new virtual network device on the other side can register itself with Barrelfish as network driver. This leads to a connection from Barrelfish over the virtual network device inside the VM, through the bridge and the original Linux network driver to the real network card and therefore to the connected network.

2.4 DMA Implementation Alternatives

A major problem is getting the Direct Memory Access to work correctly. The virtualized guest operating system only runs in a virtualized address space and usually doesn't know about it. Hence it writes virtual addresses into the device memory. Even worse, not only contain the pointers from device memory wrong physical addresses but they may also point to (in principle) arbitrarily linked structures. How this structure is actually organized is device dependent. There exist a couple of solutions to this problem which we explain in the following sections.

2.4.1 Paravirtualization

A paravirtualized (modified) guest operating system could calculate correct physical addresses. Paravirtualization would not only allow to map the memory directly but also get rid of VM exits completely, removing a considerable amount of virtualization overhead. The guest can misuse the DMA to gain access to the whole system memory, but arguably, a modified guest is also a trusted one.

2.4.2 Translation

It is possible to intercept the device memory access from the virtual guest system, inspect the write and if necessary translate the written address. Problematic are nested structures: It must be taken care of the guest having set up the complete nested structure before translation begins. However, at some point the device must be told that it can now use the structure. At this point it is safe to translate the structure, but figuring out the correct moment is device dependent. Performance is hit considerably, because intercepting the write means that the memory region cannot be mapped directly. So every guest write/read access causes a VM exit. Security can be guaranteed by checking if the translated address is valid for the guest.

2.4.3 IOMMU

Chipset manufacturer have recently begun to include a hardware unit which performs address translation between the PCI bus and the main memory, called the IOMMU. The host simply has to set up the unit as it does for normal virtualized memory. As an extra hardware unit performs the translation when needed, it is not device specific. Performance should be very good as everything can be mapped directly. Also the IOMMU allows to restrict the area of memory which gets accessed, therefore maintaining encapsulation of the guest.

2.4.4 Discussion

The IOMMU is considerably the best option, providing high performance and security. In addition, there is no need to modify the guest. The only drawback is that it must physically exist and not all systems do have one. We wanted our solution to work on computers without an IOMMU, hence leaving the choice to either go with a modified guest (paravirtualization) or let the implementation become device specific. We opted for the later.

3 Related Work

3.1 PCI Passthrough

- XEN [6] is able to passthrough PCI devices to either a paravirtualized guest or to an unmodified guest using an IOMMU.
- VirtualBox supports PCI passthrough with an experimental additional module. It makes use of the IOMMU, which also is a limiting factor since this only works on systems which contain and support an IOMMU [4].

3.2 Guest - Host Communication

- VMWare supports a special socket API for guest host communication [3].
- VirtualBox provides the VirtualBox Guest Additions which can be installed in the guest operating system. This packet contains drivers and applications for improving the performance and usability of the guest operating system. Furthermore, it establishes a generic host/guest communication channel for exchanging data between the guest and the host system. [5]

4 Implementation

4.1 Overview

Figure 2 shows the memory situation. First there is the host-physical memory in Barrelfish *H Ph* of which two Gbyte are allocated as guest-physical memory for Linux. In the same space, there is the memory mapped device memory of the Intel card shown as *ixgbe*. These two regions are also mapped into the host-virtual memory *H VM* of the process vmkitmon. Vmkitmon's task is to control the virtual machine's process and handle any instruction that traps (actually causes a VM exit). In the same memory region, the ixgbe and our virtual network adapter *VM driver* are accessible, although they are not directly mapped but its access is managed by *vmkitmon*.

4.2 Map real PCI hardware into the VM

As mentioned before, our implementation is device specific. More precisely, it is written for a PCI Express Intel 82599 10Gbe Controller. The network driver in Linux for this device is called *ixgbe*.

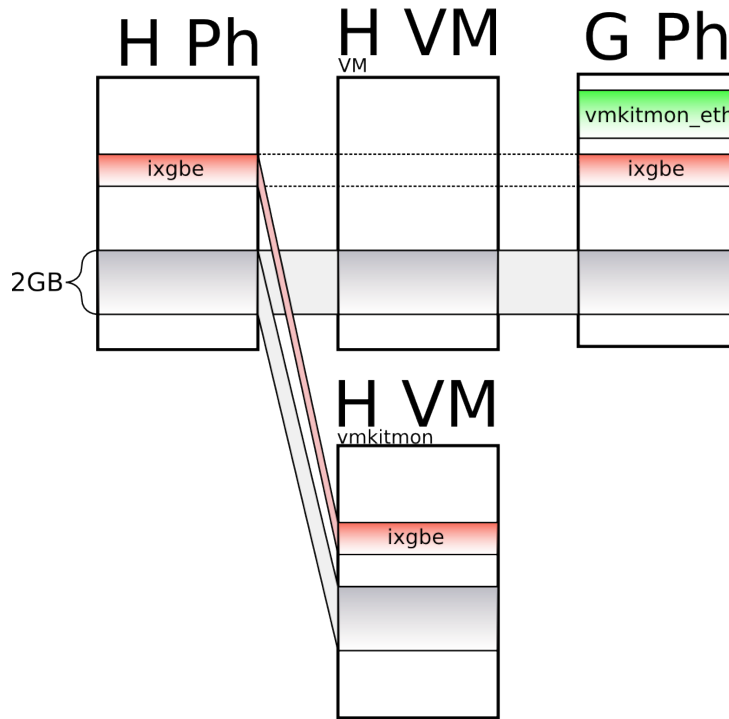

Figure 2: Memory Layout Overview

4.2.1 PCI Subsystem

On the x86 architecture the PCI bus is usually accessed over a Host-Bridge which is made available in the I/O port space. An access to this space within the VM leads to a VM exit, which is basically a context switch from the guest system to the host system or more specifically to the VM monitor which controls the virtual machine [1]. We catch these VM exits and perform corresponding actions, for example PCI configuration space read or write calls.

4.2.2 PCI Configuration Space

In order for Linux to detect the connected PCI hardware, it has to scan the PCI bus and access the PCI configuration space header of the devices. Since Barrelfish supports PCI devices, there are already procedures that access the configuration space of a PCI device given its bus and device number. However, this functions were only accessible inside the PCI module and therefore not much use for us. So we exported this functionality by introducing two new calls for the PCI service, one for read and one for write access to the configuration space. This allows any device driver that connects to the PCI service to access the configuration space of its assigned

device. Since there is a VM exit for every read and write call to the virtual PCI bus, we can intercept this action whenever the Linux inside the virtual machine tries to read from or write to the configuration space and we can call these new access procedures from the PCI library. With this method, read calls to the PCI configuration space from inside the VM are directed to the actual PCI bus and return the real value from the hardware device. Also, write calls to the PCI configuration space are directed to the actual PCI bus and the values on the hardware devices are modified.

4.2.3 Device registers and interrupts

Inside the PCI configuration header different memory mapped regions may be found. The operating system activates the memory mapped region by setting the corresponding bit in the configuration header. We don't perform any configuration of regions but rely on Barrelfish (or the BIOS) to do the work. We also don't translate anything at this level. This means that the guest operating system sees the device at its real physical location. However, the memory is not mapped into the address space of the guest but we let the access cause a VM exit and handle it there, allowing us to modify write and read operations. As we register ourselves at the Barrelfish PCI server, we get interrupts caused by the card, no matter by which mechanism they are passed (for example MSI - Message signaled interrupts). We simply pass them on to the guest. VMKit emulates a legacy *programmable interrupt controller (PIC)* (Section 6.7.1, [1]), but Linux is permissive enough and forwards the interrupt correctly to the driver.

4.2.4 Handling DMA

As mentioned earlier, handling DMA requires some device specific knowledge: Where are pointers to main memory stored and how does the memory structure look like? We find the answers in the datasheet [2]. The card communicates with the operating systems over two ringbuffers, one for transmitting packets and one for receiving. The base address of each is stored in device memory. For the head and tail of the buffer, only relative datums are used. Hence it is enough to translate the base address when it is written by the guest. Unfortunately, the ringbuffer entries do not point directly to the packet data, but point to receive/transmit descriptors. These consists of status flags and pointers to data/headers. The situation is depicted in Figure 3.

Therefore we must find the instant at which the structure, which is written by the guest, is valid. This is at the latest the case when the ringbuffer's tail is incremented. So whenever we detect a write to a ringbuffer tail, we translate every entry between the old value (which we read from device memory) and the new value (which we get as an argument of the page fault

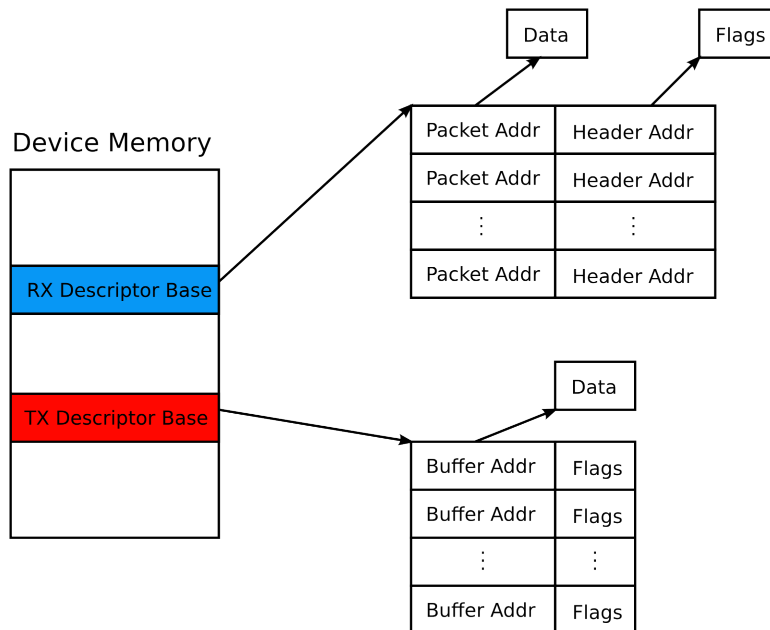

Figure 3: Intel 82599 Memory Layout

handler). This way it is guaranteed that we never miss a translation and that a translation occurs only once. The translation happens before step two in Figure 4.

4.3 Guest - Host communication

4.3.1 Setup

To establish a connection from Barrelfish into Linux (and back) we added a new virtual PCI device to the VM. It is used to transfer data between the Linux inside the VM and Barrelfish. The PCI configuration space header is preconfigured with a fictional vendor and device ID and a base address for the device registers. The counterpart of this virtual PCI device is a driver inside Linux. This driver registers itself with Barrelfish as a network driver for the specific device ID and as soon as Linux finds the PCI device, its initialization method is called. In this method the normal initialization stuff for Linux drivers happens. A data structure for the driver specific data is allocated, a network device is registered and the device registers are mapped into the address space. The interesting part begins when the device is activated (by *ifconfig eth1 up* in Linux). Then another initialization procedure is called to prepare the device for transferring data. The memory for the transfer is allocated and a special data structure is set up. More to this structure later. Since the counterpart in Barrelfish needs to read this memory region too, we need its guest-physical address. For this reason

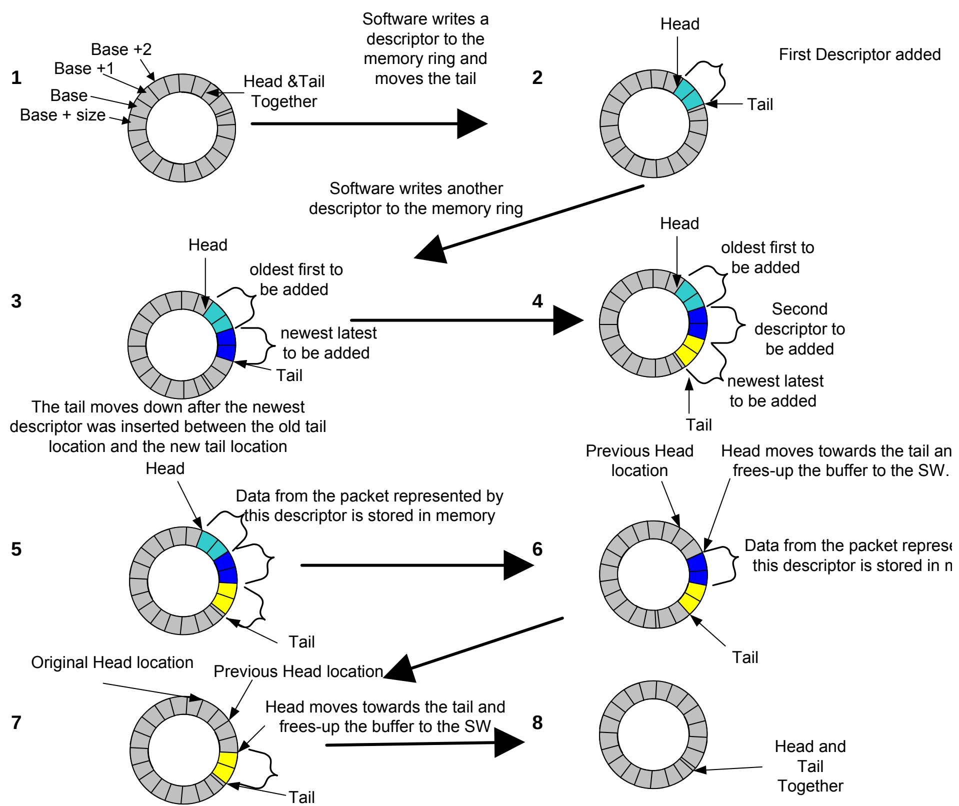

Figure 4: Operating system - Intel card communication

we allocate DMA memory which gives us the virtual as well as the guest-physical address. The guest-physical address is written to a device register, so the virtual device in Barrelfish can read the value from there. Now the virtual hardware has to be informed that the driver is ready and prepared. This happens by writing the ready bit to the control register. Now the communication channel from Barrelfish to Linux is almost complete. After the Linux driver requests the interrupts from the virtual network device, the Linux driver is ready to receive and send packets.

The idea is that this construction with VM and Linux is completely transparent for the Barrelfish network stack. Barrelfish should only see a network interface. For this reason we have to register a network driver. This driver is a normal Barrelfish network driver, gets packets to send and delivers received packets to the network stack. Under the surface there is of course no actual networking hardware but only our virtual PCI device, which exchanges data with Linux.

4.3.2 Transmit and Receive Descriptor Tables

For transmitting data from Barrelfish to Linux and vice versa, two shared data structures are used. One of them is the Transmit Descriptor Table *(TX Desc Table)*. An entry in this table contains of two 32 Bit values. The first one is an address to the transmit buffer containing the data to send, the second one is the length of this packet. A length of zero indicates an empty and therefore free buffer. The table's purpose is the transmission of Ethernet frames from Linux to Barrelfish. The Receive Descriptor Table *(RX Desc Table)* looks exactly the same, but it's purpose is the opposite direction (Barrelfish to Linux).

4.3.3 Packet transfer from Linux to Barrelfish

When the Linux network driver gets a packet to send, it first searches for a free slot in the transmit descriptor table *(TX Desc Table)* by looking for the first table entry with length equals 0. The packet is then copied to the corresponding transmit buffer and the length is set to the length of the packet. Afterward, the virtual network device is informed about the new packet by setting the transmission bit in the control register to one.

The Barrelfish counterpart on the other side scans the Transmit Descriptor Table for any valid packets (where length not 0) and delivers the found packets to Barrelfish using the Barrelfish network driver interface. It then sets the length of the delivered packets to 0, so that the slot can be reused.

4.3.4 Packet transfer from Barrelfish to Linux

The inverse direction works very similarly. The Barrelfish network driver gets a packet to send and finds the first free receive descriptor by scanning

the Receive Descriptor Table. As before, free descriptors can be found by looking for any entries with length equals 0. The driver then copies the Ethernet frame to the receive buffer corresponding to the first available table entry and sets the length according to the packet length. To inform the Linux driver about the pending packets, an interrupt in the virtual machine is triggered. In the interrupt service routine of the Linux network driver, the Receive Descriptor Table is scanned for any valid entries and the packets are copied into Linux socket buffers *(SKB)*. These buffers are then passed to the Linux network stack to be processed and in our case to be sent through the bridge to other network driver.

4.4 Bridging

Finally, to connect these two network devices in Linux, we use the Linux bridging functionality [7]. It connects two Ethernet devices on Layer two, so it is similar to a switch and hence invisible to Barrelfish.

5 Performance Evaluation

We compared our implementation to a native driver solution using two different measurements: Throughput and Latency.

5.1 Throughput

We measured the time which is necessary to fetch a file over the LAN via NFS. Figure 5 shows that our implementation is able to reach about 8 MBit/s while the native driver gets close to 100 MBit/s. Interestingly, our implementation reaches its top speed not at the largest filesize, but on files around 64 Kbyte. We don't exactly know why this happens, but we guess that at this point the NFS client divides the file read requests into more than one batch and that this somehow slows down the total throughput.

5.2 Latency

To measure the latency we set up a remote host to ping Barrelfish and recorded the time at four different steps. The first point is where an interrupt from the physical card arrives in our pass-through code, indicating that a packet has arrived. The second step is when we actually deliver the packet to Barrelfish. This timespan defines the *Net to BF* time. The third point is when Barrelfish asks our virtual device to transmit a packet. Finally, the fourth point is when the guest operating system tries to increment the *transmit descriptor tail (TDT)*. This defines the timespan *BF to Net*. The total latency is taken from the output of *ping*. Figure 6 shows how that

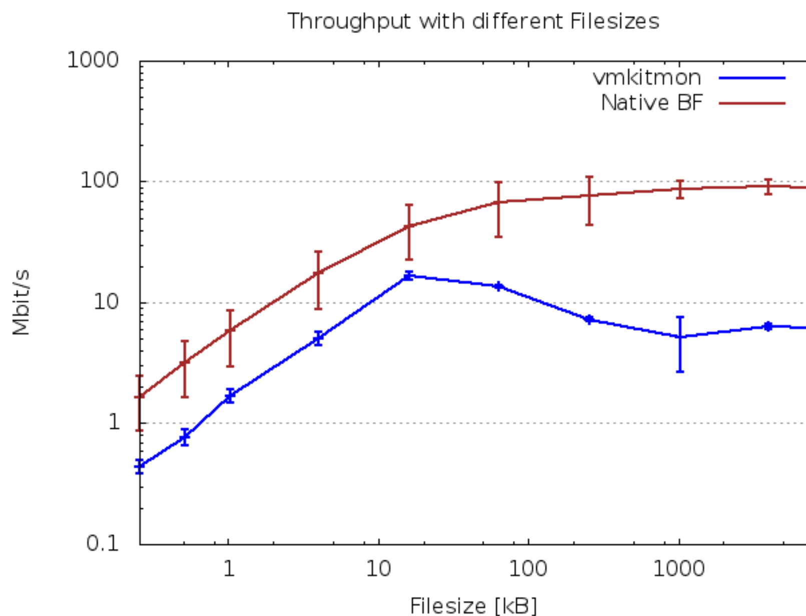

Figure 5: NFS throughput with different file sizes

the overhead our implementation adds is about 0.3ms per direction (Barrelfish to the network or from the network to Barrelfish). The size of the blue boxes should be comparable, as they measure basically the same thing. Nonetheless, the time the same step takes when routed through Linux and our implementation is considerably larger. We see three possible explanations for this behavior: First, the VM may soak up CPU usage when the ping request is processed inside Barrelfish. Second, functions from our driver may get called outside of measurement time (for example when the network stack tells us that a slot is now free again). And finally, the additional context switches into the VM may invalidate the cache so that forthcoming operations are performed more slowly.

5.3 Discussion

One fact which influences performance badly is that we copy each packet multiple times. For an incoming packet several copies happen: The Linux driver copies the data into its own DMA space, and in Barrelfish we copy again the data into a Barrelfish descriptor. Additionally, Linux (or the Bridge implementation) may perform copies. We could bring down the number of copies in our code. The Linux driver could get rid of it, if it could figure out the physical address of the packet. On the Barrelfish side, the network stack must provide an interface to fetch a packet from an arbitrary location instead of only from the network stack provided slots. Another

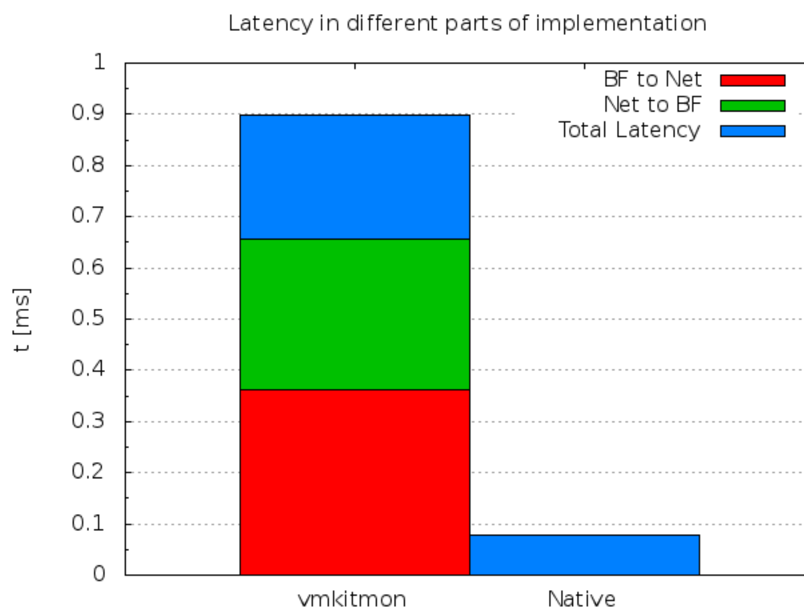

Figure 6: Latency

possibility would be to map the memory which contains the network stack provided slots into the virtual Linux memory. Then the Linux driver could perform the only copy necessary.

As we need to have control over the device memory access, we cannot map it directly. Hence for every write and read an additional (in comparison with direct memory mapping) VM exit is caused. As the Linux driver currently only increases the buffer by one element, one extra VM exit is caused for sending a packet. Also for receiving one is caused, because the buffer needs to be allocated at some point, but at least, the VM exit doesn't necessarily happen during the latency critical time between packet arrival and packet-to-Barrelfish delivery.

A more generic problem with virtualized I/O is the need for additional context switches into the virtual machine and back.

Another possible performance issue arises from the nature of bridging. Linux simply forwards all packets it receives on one of its bridge interfaces and forwards it to the other, leaving the MAC Address unchanged. To make this work, both interfaces must receive all packets, not only those for their own MAC address, so they both have to be put in promiscuous mode. Hence we perform the pass-through work not only for packets we are interested in, but on all packets which come from the wire.

6 Known Issues

Basically our implementation is tailored to the Linux driver we used. This has some consequences as our implementation left out features of the card that the driver makes no use of. These include:

- It works only with the advanced receive descriptor (the card supports two different formats, the advanced and the legacy receive descriptor)
- It works only when the Intel driver uses no more than one receive buffer and one transmit buffer
- Reading back an address is not supported (the guest sees the translated addresses)
- Although VMKit supports only 32 bit guests, the PCI card must support 64 bit addressing, as the translated address may go above the 32 bit boundary.

There are some points which work fine for the current setup, but may be problematic if one tries to run this on a different machine. Namely, the device memory address of the virtual adapter is hardcoded and there exists no mechanism to prohibit any address conflicts. Also, when trying to fetch large files over NFS, the pbuf pool gets exhausted. Furthermore, the bad performance may limit the usefulness.

7 Summary and Further Work

We have shown that in principle it is possible to use a virtualized guest operating system to provide device driver support without any additional hardware. Unfortunately the approach has some serious drawbacks: First, the implementation requires in-depth knowledge of the device to be passed through. At first we hoped that it is sufficent to provide some kind of bitmap indicating the words of the device memory which require translation. But deeply linked structures render this impossible. Secondly, the performance is not acceptable for anything except the occasional IRC conversation. Although there is room in our setting for improving the performance, we don't think it is worth the effort. It may be possible to reduce the number of packets copied in our code, but most likely the Linux kernel or the bridge performs more copies. Cutting these down would mean to modify Linux and then we'd better go with a completly paravirtualized Linux. But even more worthy than trying to improve performance by using a paravirtualization may be creating a generic passthrough with help from an IOMMU. This may not only lead to a generic implementation but may also improve performance and security. It is also questionable if driver virtualization is a

good idea at all, at least as long as we are using an open source guest operating system. Then it may be easier to just port the driver to Barrelfish. This may be simplified by providing a similar (if not completly compatible) interface to the network driver. A nice byproduct of our work is that we now have a network communication channel into the virtual machine. Also passing a device into the virtual machine may be of worth by itself. Another use case worth investigating is using the virtual machine as firewall for the host operating system.

Appendices

A System Setup

For development and testing we used a machine from the Systems Group rack with the name sbrinz1. Here is a short overview of its hardware specification and the used software.

Hardware

- CPU: 4x Quad-Core AMD Opteron(tm) Processor 8380 (Shanghai)
- RAM: 16 GB
- Network: Intel Corporation 82599EB 10-Gigabit Network Connection (and more)

Software

- Linux Kernel 2.6.37
- Linux network driver ixgbe
- bridge-utils for Linux
- Barrelfish revision 526 from 22.3.2012

B PCI configuration space

The PCI configuration space is small standardized memory region on each PCI device which can be used to identify the device, get information about it and do some configurations. It's use and structure will be briefly discussed here. The configuration space is addressed over the PCI bus and consists of a standardized header and device specific data. The header contains a class, a vendor and a device ID to uniquely identify a device. The operating system can inspect these fields and load for each device the corresponding driver. Other than that there are also some registers called Base Address Registers (short BAR) which specify how the device can be addressed and (if it supports MMIO) where the device memory is mapped. Furthermore the configuration space header tells us what interrupt line the device is connected to and many other fields, which will not be addressed here and are not very relevant for this project. The detailed structure of the configuration space header can be found in figure 7.

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Registers				10h 14h 18h 1Ch 20h 24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Cap. Pointer	34h
Reserved				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

Figure 7: PCI Configuration Space

References

[1] Raffaele Sandrini, *VMkit, A lightweight hypervisor library for Barrelfish*, Masters Thesis, Systems Group ETH Zürich, 2009

[2] Intel® 82599 10 GbE Controller Datasheet

[3] http://www.vmware.com/support/developer/vmci-sdk/

[4] http://www.virtualbox.org/manual/ch09.html#pcipassthrough

[5] http://www.virtualbox.org/manual/ch04.html

[6] http://wiki.xen.org/wiki/XenPCIpassthrough

[7] http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge