Großer Beleg

Porting of the NE2000 Device Model

Johannes Richter

December 1, 2009

TU Dresden
Faculty of Computer Science
Institute for System Architecture
Chair of Operating Systems

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Bernhard Kauer

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 1. Dezember

Johannes Richter

Acknowledgements

I would like to thank Professor Hermann Härtig for giving me the chance to work in the operating systems group. Thanks apply to Bernhard Kauer who supervised the work during the project. He assisted me with valuable remarks and helped me when I got stuck in the sources. I am very thankful to Gesine for backing me up and reminding me to find an end; my colleagues from the student-lab at TU Dresden for providing such an extraordinary good working atmosphere and especially I am thankful to Adam, my son, for reminding me that computers are not most important in live.

Abstract

A virtualization solution requires abstractions for CPU, memory and IO devices. While the former are supported by CPU extensions, virtual IO devices are in the same way crucial for full virtualization.

One way to implement them is to program these devices from scratch. But because this implementation is as time consuming as writing device drivers, there is a strong argument for reusing existing virtual IO devices. The Qemu emulator contains a large and quite stable variety of virtual IO devices, which makes a generic port of these desirable.

Another advantage of reusing the virtual devices from Qemu is that Vancouver also benefits from further development, like bug-fixes, in Qemu and vice versa.

The goal of the this report is to show that porting devices from Qemu to Vancouver in a generic way requires a relatively small amount of time (compared to rewriting) and that changes in Qemu devices are still maintainable in Vancouver.

I will present a generic adaption layer with the example of the NE2000 NIC and will examine the performance of the ported device.

Based on the evaluation of the implemented NE2000 NIC, I will present a formal model for network performance and examine several hardware features of modern Ethernet devices like for example DMA, interrupt coalescing, and switched Ethernet.

Contents

List of Figures

1 Motivation

For sharing a computer system between mutually untrusted parties it is required to apply resource isolation as well as performance isolation (Barham *et al.* , 2003). Virtualization allows to consolidate multiple physical servers in a data-center into one because it multiplexes the physical resources and implies the former properties (Microsoft, 2005; Waldspurger, 2002). Virtualization also allows server-migration, OS debugging, check pointing and a variety of other use cases.

The virtual machine monitor (VMM) provides and controls the environment for a virtual machine (VM). For full virtualization the VMM emulates the hardware interface through IO device models. These involve the complete behavior of the original devices and can therefore be quite complex. Additionally they are critical for the system's overall performance, hence they require thoughtful performance optimization. These complexities make it desirable to reuse existing device models. The Qemu emulator comprises such device models, which are already reused in the mayor existing open source VMMs (KVM, Xen, VirtualBox).

Evolving device models could turn out as a beneficial situation for all projects reusing these. Improvements in these models are only maintainable, if they are ported in such way that the basic structure remains unchanged.

In this report I will analyze, whether Qemu device models can be generically ported to the Vancouver VMM at the example of the NE2000 network interface controller (NIC).

Goals of my work are to leave the Qemu devices untouched, make the implementation as efficient as possible and provide a clear structured object-oriented interface to Vancouver.

The result will be an example implementation allowing performance measurements of this approach. Thereafter, I will develop a theoretical model of an ideal network interface for virtualization. This model will allow for hardware extensions like DMA, interrupt coalescing, and switched ethernet. The goal of this model is to account for the maximal reachable bandwidth of virtual networks and ease the selection of an network device for full virtualization.

In Chapter 2 (Foundations) I will introduce to terminology and software required in this field. Chapter 3 (Design) will explain the architecture of the adaption between Vancouver and Qemu. In Chapter 4 (Implementation) I will show the concepts of critical aspects of the implementation. Chapter 5 (Evaluation) discusses the Implementation aspects as well as the performance. To examine the performance of the adapted NE2000 device model, I will measure the bandwidth and latency between 2 VM instances. These measurements will serve as basis for the consideration of an optimal device model for virtual networks, described in Chapter 6. Chapter 7 (Conclusions) subsumes the results of this work.

2 Foundations

“Testing shows the presence, not the absence of bugs.” - Edsger W. Dijkstra

In the beginning of this chapter I will clarify the usage of the terms *emulation*, *virtualization*, *paravirtualization* and *full virtualization* in this report. Following that, I will introduce the concepts of the Qemu emulator, the Nova micro-hypervisor and the virtual machine monitor Vancouver to give a basic introduction.

2.1 Emulation

An early description of emulation is the application in the System/360 used as compatibility feature to execute code for the IBM 7074, 7080 and 7090 systems. Emulation allows executing program code not directly on its native hardware or software architecture, but in an artificial software environment (the emulator), providing the interface and protocol the program code requires. According to Tucker (1965) emulation is an interpretive technique using a combination of software and hardware. The emulator “*runs in the manner of an interpretive routine simulator program but is about 5 or even 10 times as fast as a purely software simulator (Tucker, 1965)*.” The hardware backing is therefore required to emulate code for different hardware architecture in a performant manner. So the discrimination between emulator and simulator is merely one of performance. Pure software emulation became feasible with increasing hardware performance by accepting the performance overhead.

In this report I use the term *emulation* to refere to a system (*Host*) that mimics the binary interface of the guest system. Obviously, an emulator is not restricted to run code for different hardware architectures, but may also run code for the same architecture in an artificial software environment.

Emulation can also be employed to reuse software that is not available for the used computing platform (the combination of hardware and software environment). It is also substantial for archiving software and digital documents, because of the ever-evolving technical environment (Rothenberg, 1999).

In Section 2.3, I will discuss one emulator in detail.

2.2 Virtualization

2.2.1 Formal Requirements

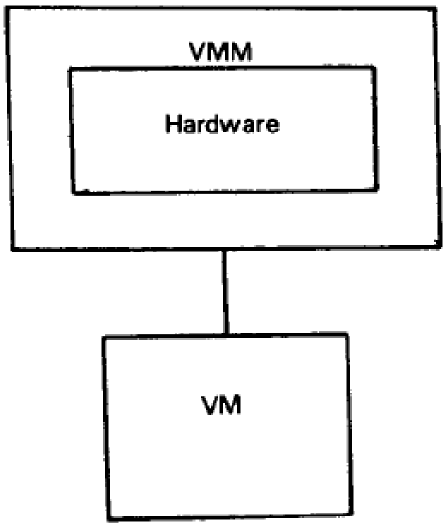

Figure 2.1: The Virtual Machine Monitor

The term *virtualization* is nowadays used as marketing buzzword with broad meaning, therefore I will give here a focused definition based on the work of Popek & Goldberg (1974).

I use the term *virtualization* as a special case of emulation with additional requirements: The software that provides the virtual machine is called virtual machine monitor. The VMM shall provide an environment identical to the original machine for programs as depicted in Figure 2.1. Programs executed in this environment shall show at worst only minor decreases in speed and the VMM shall be in complete control of system resources. The result of these requirements is that "*any program running under the VMM should exhibit the [same behavior], with exception of limited resources and differences caused by timing dependencies (Popek & Goldberg, 1974).*"

Popek and Goldberg assume that interrupts and peripheral devices, like input and output devices in a Computer (IO devices), do not exist within their definition of a virtual machine. This exception allows to emulate devices in a virtualization environment.

2.2.2 Paravirtualization

A crucial feature of current processors like X86 are protection rings that build hierarchical protection levels with defined gateways. They allow operating systems to realize fault isolation and security efficiently. When the VMM executes a guest operating system (subsequently: *Guest*) in a VM, this Guest typically calls privileged CPU instructions. This access is not possible if security and fault isolation between the VMs as VM and Host shall be guaranteed. Preventing the *Guest* from using privileged instructions circumvents this obstacle.

I use the term *paravirtualization* to describe a specific type of virtualization that permits the adaption of the guest OS to run it on a hypervisor, which provides a similar interface, but not the same as the native hardware.

This adaption effectively allows the guest OS to run without privileged instructions on top of the hosting operating system (Host) and therefore enables virtualization on architectures that do not support it for all CPU instructions (like x86 before SVM/VT).

Barham *et al.* (2003) used the paravirtualization approach within the XEN-Hypervisor to provision resource isolation. Furthermore, device drivers can be modified to access no longer the IO device itself, but a resource multiplexer provided by the hypervisor.

Paravirtualization is efficient, because it allows to use a higher level in interface abstraction than the hardware device interface. Therefore the overhead is lower than accessing virtual device models, where each access requires intervention of the hypervisor.

2.2.3 Full Virtualization

In contrast to paravirtualization full virtualization allows to run unmodified guest OS. This requires the VMM to provide the full hardware-interface. On X86 full virtualization requires a virtualization extension that traps when privileged instructions are executed from the Guest. This allows the Host introspecting and handling this instruction without having to check all unprivileged ones. Operating Systems require IO devices to communicate with the Guest. But Guest OS could not be allowed to access IO devices directly, otherwise they could easily compromise the Host. The way out is IO device emulation: the VMM emulates - using a software model (IO device model) - the behavior of an IO device when accessed from the Guest.

Full virtualization therefore bears the ability to run the most exotic proprietary legacy OS (like OS/2 or Novell Netware) as no modifications to it are required. This comes at the cost of IO device emulation, which is especially expensive for old IO devices with a low level IO hardware interface. Furthermore, an instruction emulator for real-mode code is required on Intel CPUs.

2.3 Qemu

Qemu is a portable general purpose CPU emulator and virtualizer. As an emulator, it is capable of executing applications and full operation systems on a different architecture.

Qemu emulates an elaborate set of emulated IO devices: Cirrus CLGD 54xx VGA, PS/2 mouse and keyboard, IDE hard disk, i8259 interrupt controller, NE2000 NIC, RTL8139 NIC, e1000 NIC and a lot more.

In this report, I use the distinction between Qemu device model to refer to these emulated devices and Qemu emulation code to refer to the remaining Code of the project, which implements, for example CPU instruction emulation.

The Qemu device models are also used by KVM, Xen HVM and VirtualBox. The usage by other VMMs, showing the maturity of the code and the large set of device models make them an attractive device model source.

2.3.1 IO Device Models

The devices models of the Qemu project are aggregated in C files, each of them including shared headers for general hardware access and for device specific features like ISA, PCI, network and so on. These headers define the device interface to the Qemu emulation code as well as the hardware emulation.

2.3.2 IO Device Interface

The interface to a Qemu device model is typically a bunch of static C functions implemented in a single module. All state information of a device are separated in device-state structure, which is accessed by the device-logic functions through a void pointer. This allows for encapsulation and information hiding like in object-oriented languages.

At initialization time, the device registers a set of address ranges and bus-size specific IO callback functions through the

```
register_io_port_write(int start, int length, int size,
    IOPortWriteFunc *func, void *opaque)
```

in a global IO-port table. With the final parameter of this function, it registers the device state in the table, which passes the state back to the callback functions.

As a network card also requires to send and receive network packets, it registers the functions `fd_read()` and `fd_can_read()` at the Qemu VLAN. In the NE2000 example is the function:

```
qemu_new_vlan_client(nd->vlan, ne2000receive,
    ne2000canreceive, s);
```

To send a network package the NIC calls the function `qemu_send_packet()`, which broadcasts the provided packet unconditionally to all other NICs registered to the same VLAN.

Within the Qemu devices edge- and level-triggered IRQs are signaled by calling `qemu_set_irq()`.

2.4 Nova

Nova is an experimental micro-kernel, written by Udo Steinberg. It provides virtualization support and is therefore called a micro-hypervisor. It aims to run secure applications next to unmodified commodity operating systems. In contrast to other hypervisors, it keeps the trusted computing base (TCB) small by running VMM and device drivers in user mode. The following subsections describe the Nova Architecture presented in Steinberg & Kauer (2008).

2.4.1 Hypervisor Services

Nova conforms to the micro-kernel paradigm to implement only those features in the kernel, which are not implementable in user-space due to security reasons or because they would imply significant performance penalties.

Nova is the only process running in the most privileged ring 0. The micro-kernel starts the initial system tasks like the root partition manager (sigma0) and the VMMs. The Nova micro-kernel provides memory- and cpu-time isolation as well as access permissions based on capabilities.

A process holding a capability for a certain resource is allowed to access and use it. In contrast to access control lists capabilities can be delegated and revoked. They allow a fine granular and flexible management of resources. For example can processes pass the right to access an IO port to other processes with witch, they have a communication relationship established. Furthermore, can the root partition manager exclude a sub-process from accessing an special IO port, for which it already holds a capability, by revoking its capability for that IO port. Nova implements capabilities as references to kernel objects. In general there are capabilities for IO ports, memory and portals.

Inter-process communication (IPC) is used for communication with the micro-kernel and between Applications. IPC takes place between communication endpoints, the so called portals. Threads with the capability for a portal can establish a communication relationship with the corresponding endpoint.

A client thread that holds the capability for a portal of a server e.g. is able to send this server thread a message, if the server is waiting in the portal. Implicitly the client thread sends with its request also a reply capability for the server.

2.4.2 User-Space Services

Nova provides two kinds of interfaces for applications. The first one is that of a multi server system, which is typical for a micro-kernel, while the second is that of a virtual machine. The multi server interface allows for secure applications with a small trusted computing base.

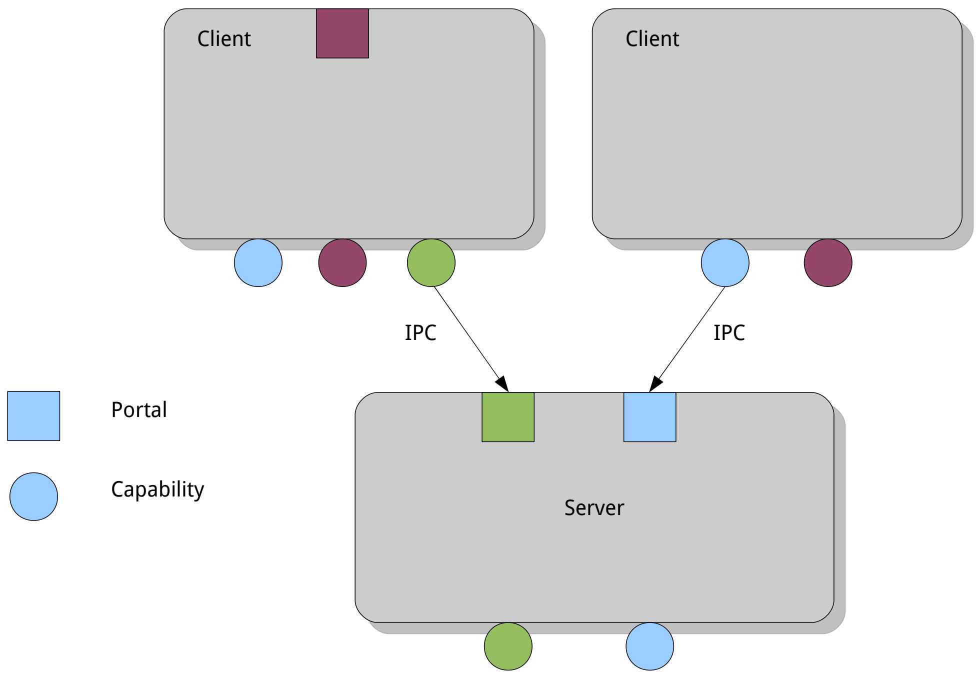

Figure 2.2: Nova Inter Process Communication

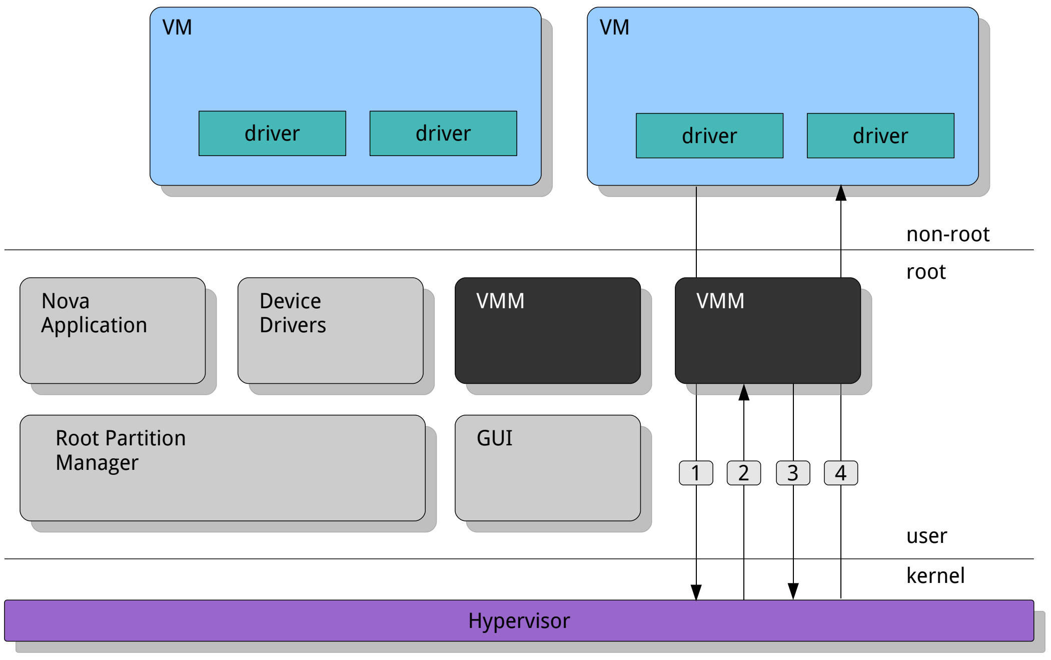

Figure 2.3: Nova Architecture

The virtual machine interface provides full virtualization support by using a virtual machine monitor running as separate user-space task, thus not increasing the TCB of multi server applications. The hypervisor allows access to specific interrupts, exceptions and #VmExit by the guest OS at predefined portals. To isolate the virtual machines from each other there is exactly one VMM instance for one VM. Therefore a fault occurring in one VMM stays isolated from other, independent VMMs. The VMM provides front-ends of virtual devices. These device models emulate the functionality of real hardware devices and can make use of real hardware devices by accessing driver applications running as separate tasks next to the VMM. The driver back-ends were written from scratch by Bernhard Kauer to ensure minimality and stability.

2.4.3 Interaction

A VMM running as native Nova Task allows the execution of unmodified guest OS by providing full virtualization. For this purpose it supplies the VM with memory and emulates privileged instructions as well as IO devices.

Figure 2.3 shows the guest OS in the VM executing a privileged instruction: The CPU traps the execution of a privileged instruction from the *Guest* and switches control from the VM to the hypervisor (1). The hypervisor sends an IPC message containing the cause for the VM-exit to the VMM (2). The VMM replies with the new VCPU state resulting from emulation of the privileged instruction or the behavior from the IO device model (3). The device emulation in the VMM can also access real hardware device drivers, which are located in separate tasks running next to the VMM. With IO-MMU support the VMM can also facilitate direct device access, while ensuring the restriction of DMA to predefined memory regions. After accomplishment of the instruction emulation the new VCPU state is injected into the VM (4) and the *Guest* resumed.

2.5 Vancouver

Vancouver is a component based virtual machine monitor programmed from scratch by Bernhard Kauer for the NOVA micro-hypervisor. The device models are designed as independent components. For each VM, one Vancouver instance is running as a user-space application on top of the Nova micro-kernel, providing isolation of the VMs. The different Vancouver instances communicate by IPC messages. For virtual network, the VMM instances requires to exchange data. This data exchange is realized by shared memory regions and semaphores between the VMM instances and sigma0 to provide security and to improve performance.

If the *Guest* accesses an IO port, the CPU traps and signals the hypervisor (NOVA) of the #VmExit. The VMM obtains an IPC from Nova (as described in 2.4.3) with the fault state and calls all virtual devices connected to a virtual bus. Each device registered to this bus has to decide if it handles the given address. When it does, it updates its internal state (set timer, modifying memory) and external behavior (triggering interrupts, changing device register). Device drivers a realized as a separate Task from the VMM. These drivers provide the back-ends for the virtual devices.

Vancouver abstains from paravirtualization to support OS in a generic manner. It is assumed that virtual devices with a highly abstract interface show a performance that is about as good as an artificial paravirtualized device. By using only device emulation, there is also no need to provide drivers for all the different operating systems running in the VMM.

2.5.1 IO Device Interface

The interface between the VMM and an IO device model is written in C++. A device model is encapsulated in a class and has to provide two functions per bus it communicates with; a write function:

```
bool write(Mword address, Mword value, TypeClassBUS_IOIO08)
```

and a read function:

```
bool read(Mword address, Mword &value, TypeClass BUS_IOIO08)
```

.

These functions are returning returns whether the access was handled by the device model, for example when the address is out of range for the device. All reads and writes to the bus are seen by any device that is registered to it. The 3rd parameter of the read/write function is the bus-type, used by the super-class StaticDevice to generate a `read/write_static` function for the corresponding bus as a performance optimization.

The resources and configuration a device uses are supplied at start-up time via the constructor.

These are for instance the IRQ bus, which is a similar abstraction like the IO-buses, the TimeSource (which allows for a notion of time, and triggering timer), the device base address, the IRQ address and device specific settings like, in case of a network interface, a MAC address.

2.6 NE2000 NIC

The NE2000 is a low-cost network interface controller based on the 8390 Ethernet chip from National Semiconductor (1996). It is designed for interfacing 8-, 16-, and 32- bit microprocessor systems.

The NE2000 NIC uses two ring buffers for packet handling. The chips DMA logic uses the receive-buffer, comprised of a series of contiguous fixed length 256 byte pages to receive and store packets.

Features of the network card are physical, multi-cast and broadcast address filtering. It provides two 16-bit DMA channels, but with a restriction to local memory.

Therefore the internal memory of 32 KByte is accessed by reading and writing to/from IO ports.

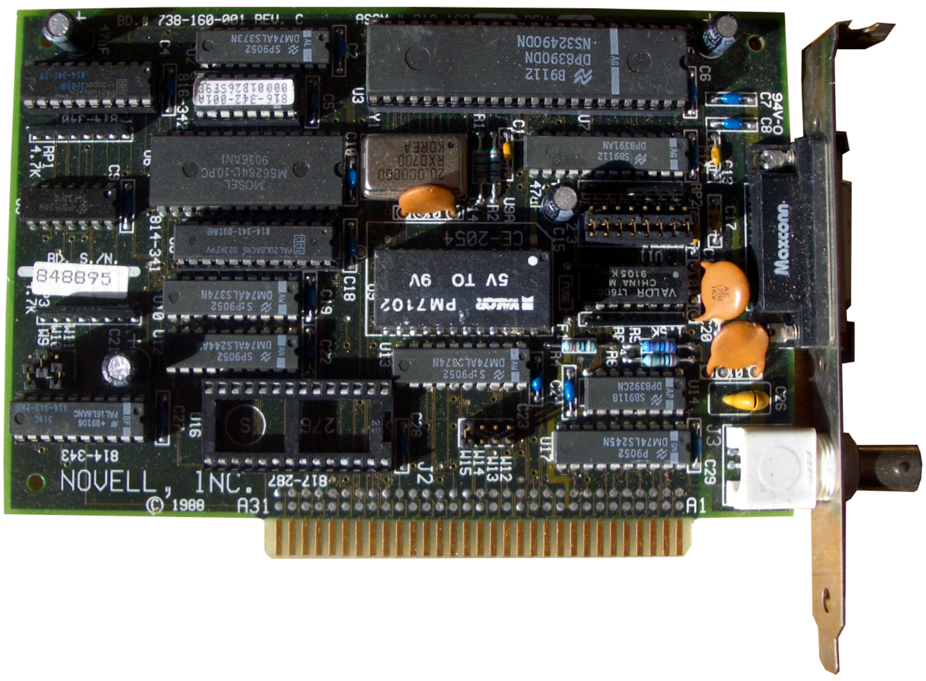

Figure 2.4: 8-bit ISA NE2000 NIC

3 Design

3.1 Design Continuum

When examining the decision space for reusing code from a different project, there are two obvious extreme cases: full reuse as black box- or as white box-approach.

In the case of virtualization, full reuse means that the Qemu emulator would be run as an unmodified system and would be accessed by the VMM for emulation of IO devices through a native interface. The benefit of this approach is the low complexity it takes to embed Qemu in a VMM. The modifications to Qemu would be to introduce new interfaces to access the IO devices from outside the application. This approach was for example chosen in KVM. Therefore, the implementation is relatively inexpensive. Also, improvements in the Qemu code would further contribute to the reused code base, because the modifications to it are small and could easily be redone. But simply running a complete Qemu for each VM instance would not only substantially increase the overhead at run-time, but most importantly the trusted computing base, which would contradict the micro-hypervisor concept.

In the other extreme of the design space, the Qemu virtual devices can be considered as a bunch of functions, which can be modified to fit in the new VMM environment. This way, the devices can be tailored exactly to the new VMM structure, which means lower complexity and therefore lower run-time overhead. The costs of this choice are high in the long run, because the maintenance for developments in Qemu would require to manually modify the ported devices.

A good compromise of those extremes is the use of an adaption layer between the VMM and the Qemu devices. This bears benefits from both former approaches: the overhead only depends on the adaption layer and should be slight. No code changes are required to the Qemu devices itself, because the adapter acts like the standard environment to them. In the VMM also no additional features are required as the Adapter provides the same interface as the other virtual devices. As with the full reuse approach, we would still benefit from improvements in the Qemu code.

Thus, I have chosen to implement an adaption layer to embed the Qemu virtual devices in Vancouver.

3.2 Architecture

3.2.1 Adaption Layer Structure

The structure of the adaptation layer is determined by the Vancouver interface at the one hand and by the Qemu interface on the other. The goals of my work are to leave the Qemu device untouched, make the layer as thin as possible and provide a clearly structured object-oriented interface to Vancouver. Furthermore, it has to bridge the gap between Qemu, written in C, and Vancouver, written in C++.

For the Vancouver interface, I chose to abstract the functionality common to all Qemu devices into a general QDevice class. For now, the specific device inherits from this class, so that only device specific functionality needs to be implemented. The specific code for the NE2000 model is fewer than 100 LOC. If the necessity to share more common code between devices of a certain class (like network or sound) should emerge in future, this design allows also the introduction of an additional device-class specific layer.

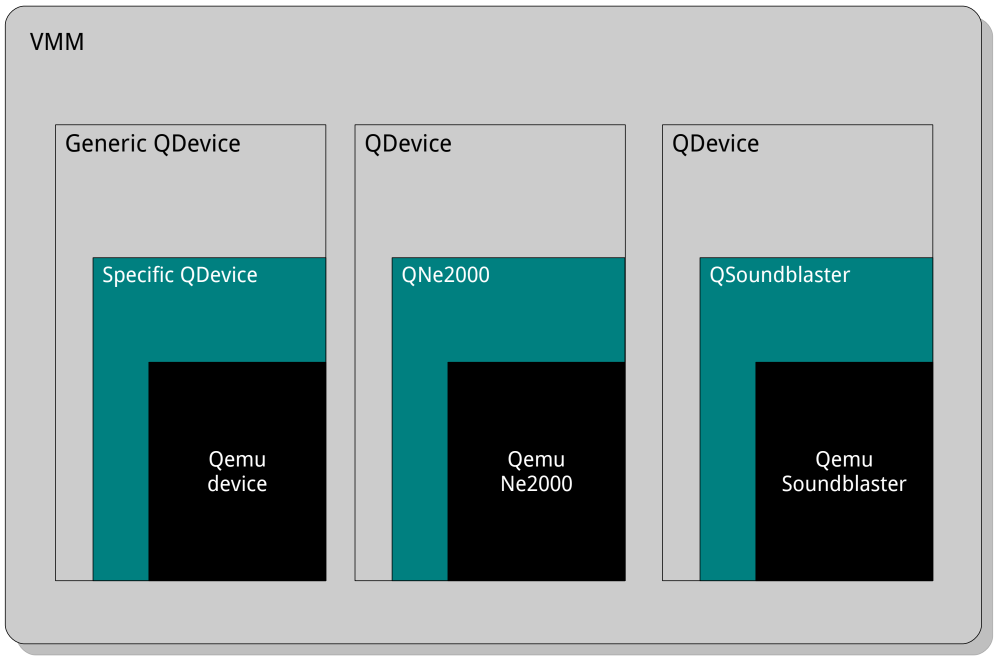

Figure 3.1: Adaption Layer Structure

3.2.2 Generic Device Abstraction: QDevice

QDevice is accessed through the interface described in 2.5.1 Vancouver IO device interface with its write and read functions. All calls to QDevice's IO functions are translated to IO functions of the Qemu device. For this purpose, the Qemu device registers its IO functions to a device specific translation table.

The constructor for a generic device `QDevice(bus_irq, base, irq)` is called for each specialized device and sets up a generic IO translation table, as well as an IRQ handler, which forwards interrupts to the Vancouver IRQ bus.

3.2.3 Specific Device Abstraction: QNe2000

The custom device layer automatically uses the functionality of the generic one by inheritance. Additional functionality is encapsulated in the custom device class. This class calls the initialization function of the Qemu device and provides it with resources: In the case of the NE2000 NIC these resources are the base address and the NICInfo structure, which holds for instance the MAC address for the device, and the VLAN, to which it is connected.

Figure 3.2 shows the processing of a guest IO using the example of the NE2000 model:

1. Access to a virtual device through an IO port causes a VM exit.
2. The hypervisor synthesizes an IPC message that comprises the fault state and sends it to the VMM.
3. The VMM accesses the specific Qemu device through the generic QDevice interface, translating the request to the Qemu device model.
4. After emulating the behavior of the virtual device, the VMM responds with an IPC message containing the new state and optionally an interrupt to inject into the VM.
5. The hypervisor injects the new state and resumes the VM.

3.3 NE2000 Model

The simplest NIC model to implement is the NE2000 because it is an ISA bus card and requires only few supporting hardware emulation mechanisms. In this section, I will describe the transmission of a network packet at the NE2000 device model and will show the shortcomings why this network interface controller is not suitable for a virtual network architecture. In the Chapter 6, I will discuss several optimization options to circumvent these shortcomings.

To send a message, an application in the VM triggers a package transmission. For that purpose, it executes the `send(sockfd, buf, len, flags)` syscall. The NIC driver copies the packet with the `rep outs` CPU instruction to the IO-port-mapped device memory.

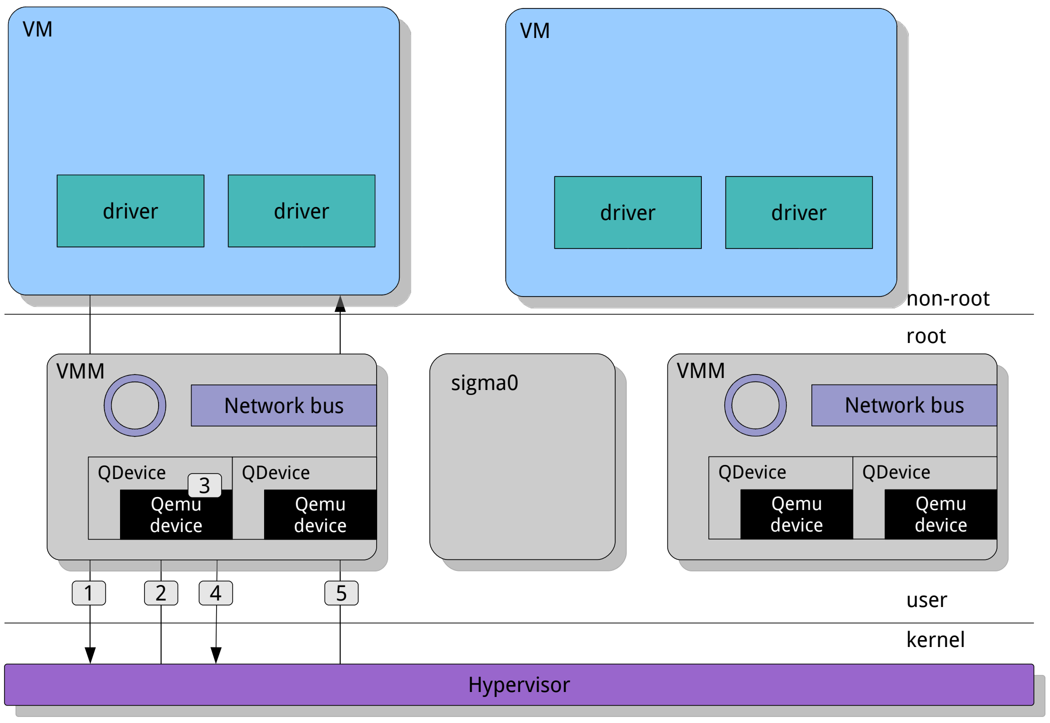

Figure 3.2: IO port access

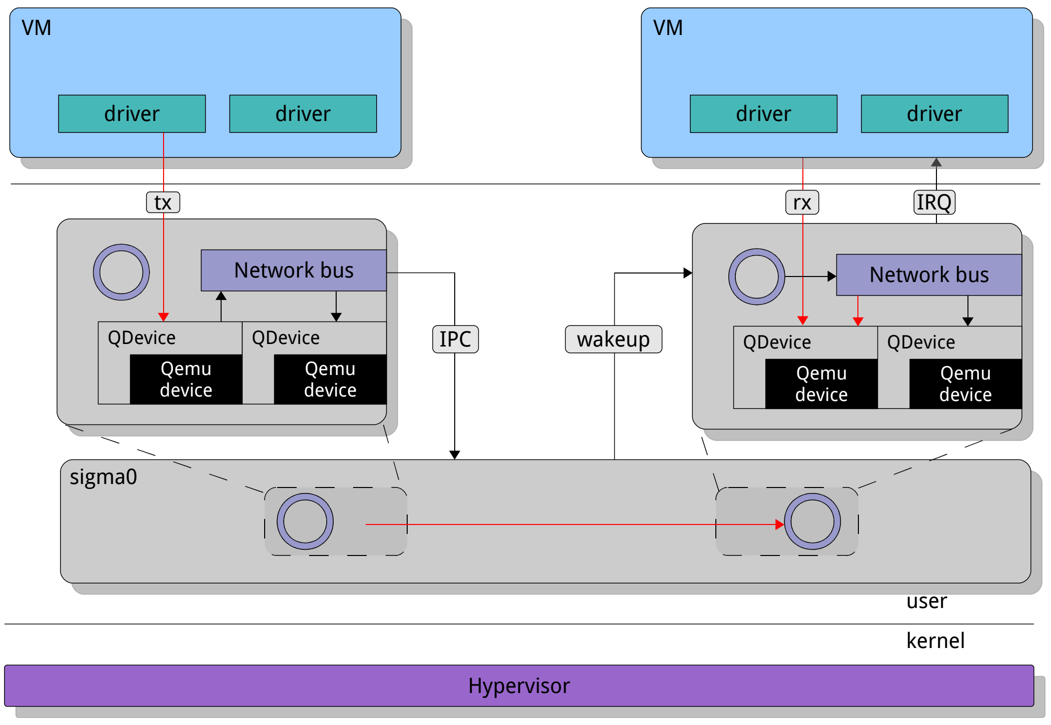

Figure 3.3: NE2000 Network Model

The CPU traps at this instruction and the hypervisor (Nova) notifies the VMM with an IPC of the failure state. The VMM emulates the instruction included in the VCPU state and calls the IO function of the Qemu device through the adaption layer, with address and data of the IO port access. The Qemu NE2000 device stores the data into its internal memory ring buffer. When the driver sends a transmit IO instruction, the virtual NIC sends the address and length of its packet buffer to the network bus.

The VMM calls all registered VMM-local receive functions of the device model and acknowledges the transmission with an TX interrupt to the sending VM. The NIC model sends an IPC with the address and the buffer size to Sigma0.

Sigma0 copies this buffer to all registered (attached) receive ring buffers of other VM's and wakes up the receiver threads in the other VMM's by a `sem_up()` syscall.

The receiver thread broadcasts the package to the receive functions of VM-local NICs. Each NIC checks the MAC address of the package and if it is addressed to the same NIC, the package is copied from the ring-buffer to the the internal memory buffer. On packet reception, the NIC signals an IRQ to the VMM, which injects it into the VM. The network card driver notified by the IRQ copies the packet with the `rep ins` into the VM memory. The performance relevant-operations for the transfer of one packet are listed in Table 3.1 .

#	Action	General Costs
1	`rep outs` to device	VmExit + memcpy(packetsize) + packetsize/bus_width * emulation(`rep outs`)
2	transmit instruction on IO port	VmExit
3	TX Interrupt to VM	IRQ
4	Bus notifies sigma0	IPC
5	Sigma0 copies packet to all VMs	(VM-1)* memcpy(packetsize)
6	Sigma0 notifies VMs	(VM-1) * sem_up
7	if the MAC address is that of the NIC: Device copies packet to internal memory	memcpy(packetsize)
8	RX Interrupt to VM	IRQ
9	`rep ins` to driver	VmExit + memcpy(packetsize) + packetsize/bus_width * emulation(`rep ins`)

Table 3.1: Packet transfer costs for IO port data transfer

Instruction	**Costs:** **Core I7-920 2.66GHz**
VmExit	1000
memcpy	12.6 GB/s
IPC	500
Sem_up	250
IRQ	2 * VmExit
emulation of `rep outs` and `rep ins`	100

Table 3.2: Instruction costs

4 Implementation

In this section, I will first explain the goals for the implementation. Furthermore, I will provide the crucial implementation details.

Maintainability is one of the most important aspects for the implementation. Therefore I strictly separated Qemu code, modified Qemu code and my own adaption layer. This structure allows to update the Qemu code as well as improving Qemu itself, when bugs have been found and fixed. Further goals consist in keeping the used code as small as possible and in achieving low run-time overhead for the adapter.

4.1 Qemu Interface Functions

To keep the Qemu device model unmodified, I had to provide the interface described in Section 2.3.2. This was achieved by including Qemu header files with minor modifications.

Adapter functions, with *C* naming convention, to Vancouver provide the functionality for these definitions. These functions are in general forwarding wrappers for the corresponding functionality. Two examples are somewhat more complex:

```
register_ioport_read/write()
```

and

```
qemu_set_irq(qemu_irq irq, int level)
```

The first function implements a mechanism to find the corresponding object and is described in Section 4.2.

IRQ handling could be completely reused from Qemu. It uses an IRQState structure, which is handed over at initialization time to the Qemu device model. This structure contains a IRQ handler and an opaque pointer.

The Qemu device model calls the interrupt handler with the Vancouver IRQ bus as opaque pointer. The handler itself is called back from the device model with the opaque pointer, the interrupt number and level. The Qemu device model calls the interrupt-handler with the Vancouver IRQ bus as opaque pointer. The handler writes to the IRQ-Bus, at the address specified by the IRQ number.

```
extern "C" void irqhandler(void *opaque, int irq, int level){
    Bus * busirq = (Bus*) opaque;
    //raise: level = 1; lower: level = 0
    if(0 == level)
        busirq->write(irq, Busirqlines::DEASSERTIRQ);
    else
        busirq->write(irq, Busirqlines::ASSERTIRQ);
}
```

4.2 IO Port Table Adaption

Because Qemu uses one big static table of IO function pointers for all devices in common, which could not be used in a component based system, it had to be distributed over the particular QDevice devices.

Each QDevice keeps four ordered fixed-size tables for different bus sizes and for the opaque structure. These store port ranges and corresponding IO port function pointer respectively opaque-structures. The look-up performance is justifiable as an access occurs in linear time and the number of elements is generally lower then 10.

Due to breaking up this static table, it is now encapsulated in the QDevice class. As the Qemu device model has to access the corresponding adapter object at run time, the adapted object writes a self reference to a global static variable `current_device`, before initializing the Qemu device model. This reference is used by the registration wrapper functions to access the matching QDevice object.

```
int register_ioport_read(int start, int length, int size,
    IOPortReadFunc *func, void *opaque){

    return current_device->register_ioport_read(start,
        length, size, func, opaque);
}
```

4.3 Network Abstraction

In contrast to the Qemu VLAN, where individual virtual network hubs simply broadcast packets to all registered receive functions using their address, this registration is not possible with the Vancouver design due to separate address spaces for the different VMM instances.

Like in the Qemu VLAN, the network consists of a bus for broadcasting packets within the same VMM. Additionally, there is a ring buffer, controlled by a network thread, in each VMM.

5 Evaluation

5.1 Implementation Costs

The implementation costs of device models for a virtual machine monitor are quite high. They can be compared to device driver implementation. These high costs are caused by complicated protocols, timing constraints in the device models, low level interfaces and, most importantly, by unavailable specifications from the device vendors.

Therefore, device model implementation is costly and a shortcut in development time is quite desirable. With my approach described in Section 3.2.1, I could show that an adaption-layer, which also encapsulates the Qemu device models in separate Vancouver models, can be implemented using very few lines of code (see Table 5.1). The contributed code can still be abbreviated by cutting superfluous definitions from the header files, but this cleaning work is not useful until the required devices are identified. At time of writing the spare headers are the biggest part of the contributed code.

The overall code size for the Qemu device models is quite big, but it also contains models for various platforms and redundant devices for each device class like network, sound, graphic, etc, which are not necessary for a small x86 VMM. I would estimate the final contributed code size required from Qemu of about 1000 LOC for headers and device infrastructure additional to the core device models. I estimate the final size for all from Qemu contributed code at under 30.000 LOC, but this is strongly dependent on how many device models are ported.

Software	**Language**	**LOC**
Vancouver	C++	11.320
Qemu	C	369.480
Qemu/hardware (models)	C	124.718
QDevice/adapter	C/C++	973
QDevice/contrib	C	17.954
Qdevice/modified contrib	C	185

Table 5.1: Code sizes

5.2 Performance

The performance of the virtual network was measured on a P4 running at 3GHz hosting two VMs. Each VM was running Linux and accessing a NE2000 device model in the VMM.

TCP/UPD bandwidth and latency where measured five times with *bw_tcp* respectively *lat_tcp* from the lmbench suite and with a data volume of 10 MB. The error bars in the corresponding graphs show the minimal, maximal and average measured value. Table 5.2 shows the average of the 5 rounds for each measurement.

Measurement	**Value**
Average ping time: (800 byte packets)	$4.39ms$
Minimal ping time:(800 byte packets; packet 2 of 3)	$3.17ms$
Average TCP latency: (800 byte packets)	$3738.14\mu s$
Bandwidth: (800 byte packets)	$0.558MB/s$

Table 5.2: Performance Measurements

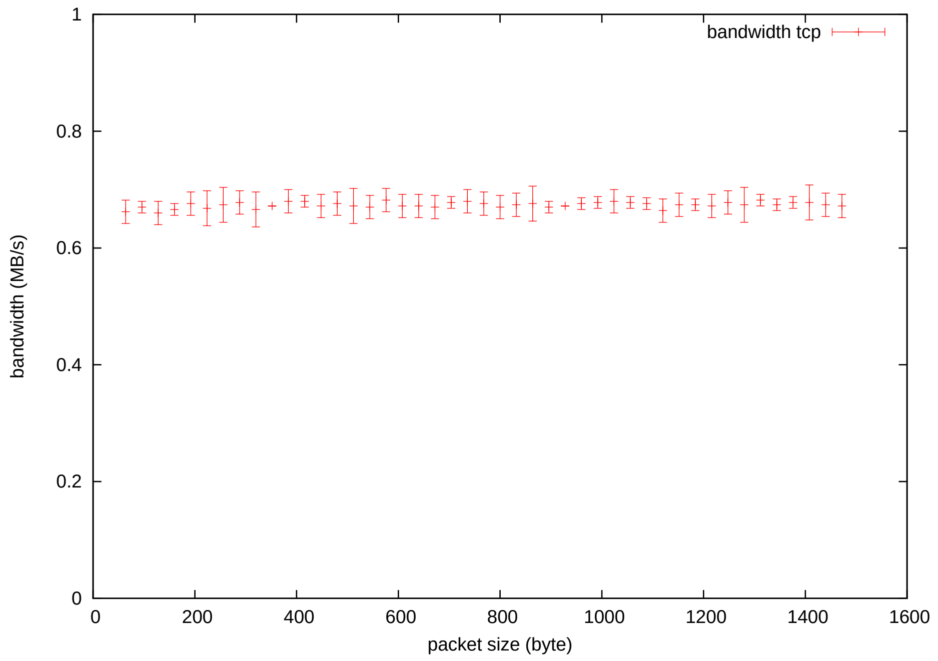

Figure 5.1: TCP bandwidth over 5 measurements

The bandwidth between the two Linux instances was measured with packet size increasing in steps of 32 byte. Over all packet sizes the bandwidth remains nearly constant at about 0.67 MB/s. I would have expected a steeply increasing bandwidth, as the per-byte costs should be negligible compared to the per-packet costs. Figure 5.1 shows that the per-byte costs are clearly dominant over the per-packet costs. The underestimated costs for instruction emulation of the `rep outs` and `rep ins` are a possible explanation for this behavior.

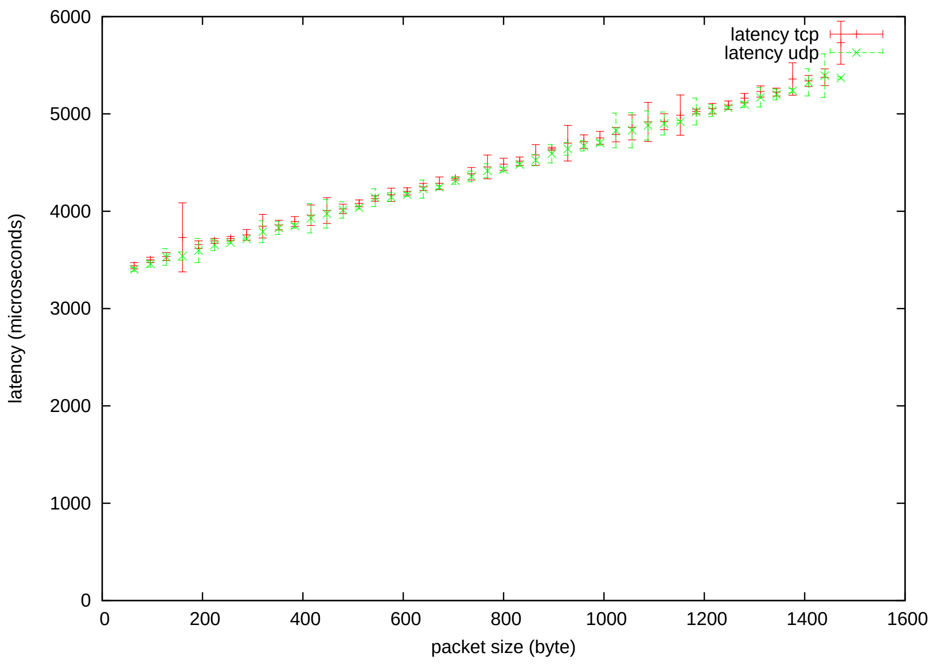

Figure 5.2: TCP/UDP latency over 5 measurements

The latency for UDP and TCP packets increases from 3439μs for a 64 byte packet to 5731μs for a 1500 byte packet, as shown in Figure 5.2. Evaluating the linear graph shows the minimal latency is about 3350μs, which is about 1000 times slower than the calculated. The scheduling of VM has a serious influence on the packet latency and was not considered in the model. The gradient of about 1.64 shows also the dominant per-byte influence in packet transmission, which should be about 1 if the latency was independent of the packet-size.

Ping times for differing packet sizes where measured with the Linux ping application. Each run sent out three pings and measured the answer time. The average ping time is about 500μs higher then the corresponding latency. This effect is a result from the high latency of the first packet as shown in Figure 5.3. Latencies of packet 2 and 3 in the series

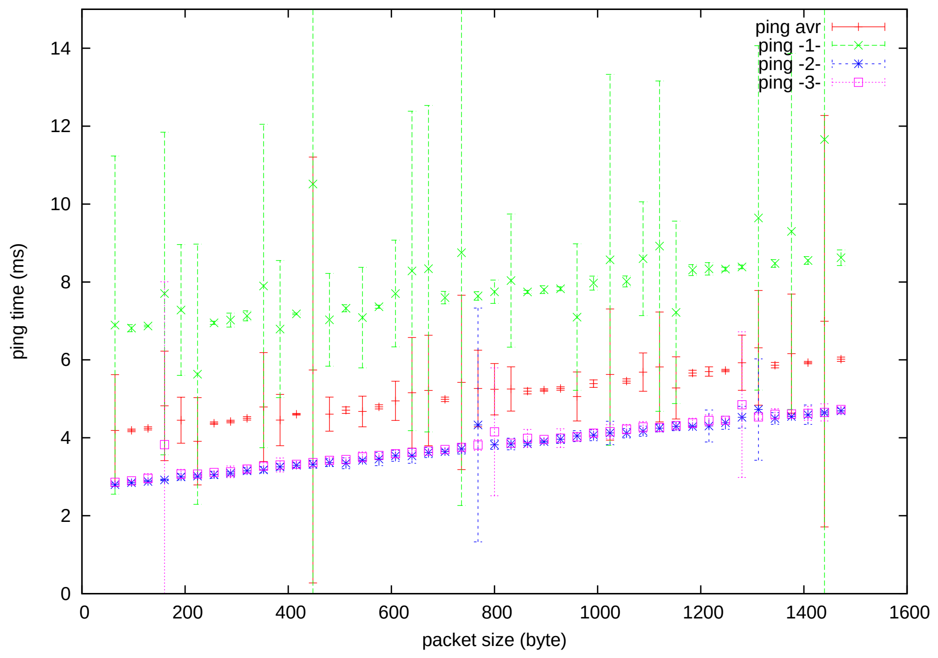

Figure 5.3: Ping time over 5 measurements

conform the measured latencies measured with *lat_tcp*. The high variations of about +/- 10ms in the ping time for the first packet indicate that the VM scheduling delays the first packet.

Ping in flooding mode sends out ping packets as fast as packets are returned, but at least 100 packets/second. Thus, it effectively gives an indication of the network bandwidth.

Figure 5.4 depicts ping response times with packet sizes increasing in steps of 32 bit. As in the *bw_tcp* test, it shows a constant average for the flood ping times with variations of about +- 6ms. The variations indicate the scheduling influence at the latency.

Compared to the measurements of Molnar (2007), we achieved ~23% of Qemu bandwidth (2.84 MB/s) and ~9% of KVM bandwidth (7.41 MB/s). As Molnar measured bandwidths on an unknown architecture, on the NE2000 compatible RTL-8029 and between Host and Guest, these values cannot directly be compared, but the can serve roughly as an lower limit.

The preceding evaluation shows that the minimal latency is much higher than calculated. One reason for that high latency is that the used Nova kernel delayed all IRQ's till the next hardware timer IRQ was triggered, which occurs in intervals of 1 ms. This interval accounts for 2ms of the packet latency.

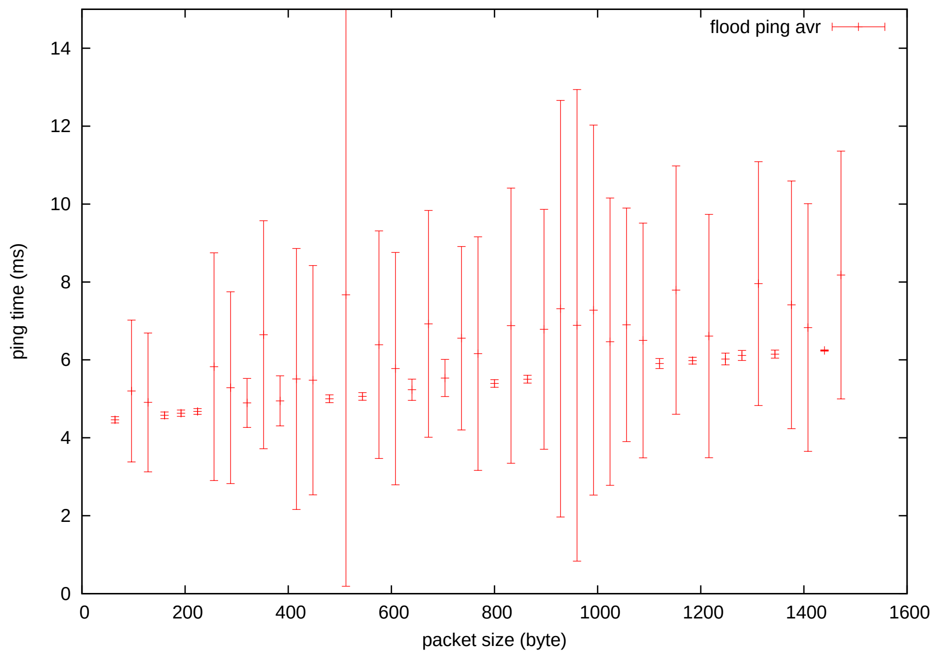

Figure 5.4: Flood Ping over 5 measurements

Additionally the the latency depends on the time-slice duration of 10ms, which also causes the variations of +/- 10ms on the ping times. The constant bandwidth for different packet sizes show that the costs per byte have greater influence at overall costs than the costs per packet.

6 Optimal Network Design

The performance evaluation in Section 5.2 shows a completely unsatisfying bandwidth of the NE2000 device model. Among others, this is a result inherent to the low-cost NIC design. In the Section 6.1 I will explain the execution overhead of this NIC in a virtualized environment. In the sections after that, I will analyze hardware extensions of modern network devices to avoid the performance drawbacks of the NE2000. In the end of this Chapter I will consider a theoretical NIC model optimal for virtual networks.

6.1 NE2000 Shortcomings

Each packet, which is send from the OS by calling `send(sockfd, buf, len, flags)` must be copied by the NE2000 driver to the NIC's internal memory. For that purpose the driver writes each byte of the packet with the `rep outs` instruction to the IO port of the NIC. This operation requires a world switch, which is quite costly in the sense of CPU cycles and additionally the VMM has to emulate the instruction. On the receiving side, the same costs apply to the `repins` instruction, which is executed from the NIC driver in the second VM. Both of these factors, the instruction emulation and the world switch, induce enormous per-byte costs for the sending and receiving of network packets.

For each transmitted and each received network packet, the NIC device model reports an IRQ to the VM. An IRQ causes 2 VM exits. The first is necessary to inject the interrupt into the VM and a second exit occurs when the device driver acknowledges the IRQ at the interrupt controller with an EOI (end of interrupt).

As we see from the Table 3.2, the most expensive operations are VM exit, the instruction emulation of `rep outs` and `rep ins`; and the IRQs. The main goal should be to reduce the huge per-byte costs and per-packet costs.

6.2 Formal Model

In this section, I will follow a more formal approach of examining the network performance. The following symbols are used in the quantitative consideration of the virtual network performance:

- C_O^X... Costs in CPU cycles for the operation O under Condition X

- C_{in}^{IO}... describes the term C_{in} in the C_{tr} formula under the condition of IO port data transfer
- $bytes$... number of bytes to be transmitted
- $coalesced$... number of coalesced IRQs; $max(1, n_{coalesced})$

The latency in a packet-switched network is the time required to transmit the packet's data from the sending VM to the receiving VM. Thus in the case of a virtual network this is the cost in CPU cycles required to transmit the packet divided by the the CPU frequency.

$$latency = \frac{costs_{tr}(bytes)}{rate_{CPU}} \tag{6.1}$$

The resulting bandwidth is determined by the time it takes to transmit a certain amount of data.

$$bandwidth = \frac{bytes}{latency} \tag{6.2}$$

$$bandwidth = \frac{bytes * rate_{CPU}}{costs_{tr}(bytes)} \tag{6.3}$$

The costs to copy a certain amount of data in memory is dependent on the memory-bandwidth and the CPU frequency.

$$memcpy(bytes) = \frac{bytes * rate_{CPU}}{bandwidth_{mem}} \tag{6.4}$$

The general costs for transmission of a data packet in a virtual network is the sum of the cost to transfer the packet from the VM into the NIC device model, the cost to transmit it into the receiving VM's NIC model and the costs to read it from the receiving NIC model into the receiving VM.

The size of the packet must be either lower than the MTU in the general case or lower than the biggest supported chunk of data supported by the NIC model in cases of jumbo frames or TSO.

$$C_{tr}(bytes) = C_{in}(bytes) + C_{vlan}(bytes) + C_{out}(bytes) \tag{6.5}$$

The cost for data transmitting a single packet over the virtual network is the sum of the costs of the following steps:

- invoking one IPC
- copying the data to each VM's receive buffer and notifying it with an semaphore up
- copying of the packet into the destination NIC model's receive buffer if the MAC addresses match

$$C_{vlan} = IPC + (N(VM) - 1)(memcpy(bytes) + sem\_up) + memcpy(bytes) \tag{6.6}$$

The following two formulas quantify the case where packets are transmitted to and from the NIC model by copying the packet using IO port instructions. The overall data to be transmitted is segmented into packets smaller than the maximal packet size; for each of these packets, the following costs in CPU cycles apply. The Table 3.1 shows the primitive operations used in these formulas.

$$C_{in}^{IO} = \frac{bytes}{packetsize}(2 * \#VmExit + IRQ + \frac{packetsize}{buswidth} * C_{emulation}(\#rep\,outs)) \tag{6.7}$$

$$C_{out}^{IO} = \frac{bytes}{packetsize}(\#VmExit + IRQ + \frac{packetsize}{buswidth} * C_{emulation}(\#rep\,ins)) \tag{6.8}$$

Under the assumption that direct memory access is used, the terms for 1 VmExit and the emulation costs in the formula above can be reduced to the memcpy function.

$$C_{in}^{DMA} = \frac{data}{packetsize}(\#VmExit + \frac{IRQ}{coalesced}) \tag{6.9}$$

$$C_{out}^{DMA} = \frac{data}{packetsize}(memcpy(bytes) + \frac{IRQ}{coalesced}) \tag{6.10}$$

The following calculations are for the Core i7 architecture with 12.6GB/s memory bandwidth, a CPU frequency of 2.67GHz and 2 VMs. Emulation costs are assumed to be 100 cycles/byte.

$$costs_{tr} = 4000 + 2 * 1500(\frac{100}{2} + \frac{2.67GHz}{12,6\,\mathrm{GB/s}}) + 750 + 3000 \tag{6.11}$$

$$latency_{packet} \approx \frac{158386\,cycles}{2.67\,GHz} \approx 0.0593\,ms \tag{6.12}$$

$$bandwidth = \frac{1500\,byte * 2.67\,GHz}{158386} \approx 24.11\,\mathrm{MByte/s} \approx 192.92\,\mathrm{Mbit/s} \tag{6.13}$$

Assumptions:

1. data amount is a multiple of packet-size
2. C_{in}^{DMA} requires one send instruction per packet; this is not valid for TSO
3. Overhead in TCP-stack is neglected
4. the model considers only the raw performance per packet

6.3 Optimal Network Design

As we can see in Section 6.1, the biggest performance issues for virtual network performance are the world switches between the VM and the VMM, which are caused by IO port access and IRQs, the instruction emulation for `rep ins` and `rep outs` and in general copying of data. In an optimal network design, `#VmExit` should therefore be executed as scarcely as possible.

The cost per transmitted byte is directly correlated to the cost per packet, because the transmission of data volumes greater than the maximal packet size is always fragmented into several packets. So the overhead per packet can be broken down to data overhead if we assume, that the maximal packet size is used (which will not hold for very short messages). The communication scenario I will describe here is the communication between different VM's on the same Host. Additional costs apply when the communication takes place through a physical NIC, because an additional mapping to the provided features of the physical NIC in the VMM may be required Menon *et al.* (2006). The following sections will provide an overview which features of a high-level interface could be useful to reduce the network virtualization overhead occurring in the NE2000 design.

6.3.1 Data Access by DMA

The biggest overhead of the Ne2000 design for virtualization results from copying the packet copying to and from the NIC model using IO port instructions. The simplest solution is to implement a NIC capable of DMA. DMA bypasses the CPU for data copy operation and lets the device itself copy to and from the system memory.

To transmit a packet through a DMA-capable NIC, the network driver of the sending NIC registers a guest physical region for DMA transfer to the device. When the OS has locked the memory page, in which the buffer of the packet resides, the NIC can send the packet directly from this buffer without copying it. This saves one VmExit per packet for the sending NIC.

At the receiving side, the network driver also registers a receive buffer into which the packet can be directly stored from the NIC when it is received. In the reception case, DMA transfer also saves one VmExit.

The VMM has to translate the guest physical addresses of the send and receive buffer to host virtual ones, because obviously the guest physical addresses of the virtual machines may be different. Also when the VMM copies the buffers it has to deal with its own virtual addresses as the mapping between guest physical and host virtual addresses is only at the VMM's disposal. The formal description for the costs of transmission and reception can be found in Section 6.2.

6.3.2 Coalesced Interrupts

The data transfer from and to the NIC can be accomplished nearly without any CPU interaction by DMA, but with the ever-growing bandwidth of ethernet cards and a fixed packet size, the IRQ rate also increases. The interrupt-handling costs for that high IRQ rate imposes a significant overhead to the CPU and can in the extreme case lead to a live-lock at the receiving side (Zec *et al.* , 2002). Therefore, at a certain threshold the CPU is saturated with handling interrupts. One counter-measure can be to coalesce multiple interrupts. There are two distinct flavors of this principle: generic IRQ coalescing can be realized completely in software, while hardware supported IRQ coalescing requires timers in the NIC.

The following steps occur when an ethernet frame is transmitted or received in the NIC:

1. NIC generates IRQ to notify the CPU
2. CPU suspends its operation to handle interrupt
    a) context switch
3. accessing interrupt controller to determine interrupt service routine (ISR)
4. executing the ISR
5. updating interrupt counters

Generic IRQ coalescing delays the generation of IRQs (IRQ mitigation). The NIC driver acknowledges for that purpose a series of interrupts at the interrupt controller without executing the ISR, which reduces the execution of steps 2 - 4. After the predefined delay has timed out, the driver handles multiple ethernet frames in a single ISR. Zec *et al.* (2002) experimented with the FreeBSD fxp network driver and argued that the IRQ coalescing introduces additional variable delay for frame processing, which has a significant influence on the possible TCP throughput. They propose a model that can be used to determine the optimal delay, critical for TCP throughput, especially for routing network nodes.

The other place where interrupts can be coalesced is within the NIC. For that purpose, the NIC is equipped with a series of timers, that delay the interrupt generation within the NIC. During the delay, the NIC waits for other packets to arrive or to be send. Using this hardware support, the CPU is also saved from excessive scheduling and execution overhead, but the additional delay increases the average latency as well. So there again is the trade-off between latency and efficiency in IRQ processing. Short coalescing delays would be desirable for low traffic rates, while larger delay would increase the efficiency for high traffic rates. The goal for the implementation of IRQ coalescing is obviously to add no significant delay to the packet latency.

According to Intel (2007), their GbitE controllers have 5 timers to deal with the introduced latency. There are 2 absolute timers (for transmission and reception) that generally delay interrupts. Furthermore, there are 2 packet timers that trigger interrupts after a period of inactivity and a master timer throttling all interrupts and enforce an upper bound at the IRQ rate. The combination off those timers provides good results for both traffic conditions (Intel® state 11% reduction of CPU utilization for receiving and 30% for transmission).

6.3.3 TCP Segmentation Offloading

The goal to reduce the overhead per packet was tackled by reducing the number of IRQs in the previous Section: 6.3.2. TCP segmentation offloading (TSO) advances one step further: it relocates the interface between TCP stack and NIC. With TSO, the NIC is fed with data buffers bigger than the MTU and control information instead of IP packets. The effort to segment the data into packets is left to then NIC. Therefore, the CPU is saved from the segmentation effort and at the the same time allowed to feed the NIC with bigger data chunks. This reduces the overhead associated with the transmission of large buffers (for example files).

The high level interface (HLI) introduced with TSO is especially useful for virtual networks between VMs, because the device emulation can transmit the bigger data chunks provided by the NIC driver directly without segmentation, which reduces overhead in the device model. Also, big data chunks can be handled more effectively than multiple smaller ones, for example when they are mapped to an other VM with zero copy mechanisms. Menon *et al.* (2006) could show, that the introduction of a high level interface to the NIC consisting of TSO, TCP checksum offload and scatter gather DMA could improve the throughput from the *Guest* to an external physical host by a factor of 4.4, while it reduced the execution costs in the Guest by a factor of around 4. It also saved execution overhead in the VMM due to the reduction of byte processing overhead (through larger packets) and avoiding data copies. An additional requirement for the virtual network infrastructure is that is has to be able to handle the data chunks of size greater than the MTU. To transmit these data chunks to a physical interface, the VMM must map the HLI to the interface of the physical NIC. If these are identical, no further overhead is required, but if the physical interface does not support TSO, it has to be emulated in software. Another crucial benefit compared to IRQ coalescing is that TSO avoid does not introduce any additional latencies. Unfortunately, TSO is only reduces the packet overhead for the sender.

6.3.4 Jumbo Frames

One of the most simplistic approaches to reduce the overhead per packet is to increase the packet size (Dykstra, 1999). For compliance to the ethernet IEEE 802.3 standard the packet size is limited to 1500 bytes. Jumbo frames extend this limitation to 9000 byte. Studies from Claffy *et al.* (1998) could show that 50% more throughput requiring 50% less CPU load could be achieved with jumbo frames on gigabit ethernet.

Mathis *et al.* (1997) presented a model for an upper bound of TCP throughput:

$$Througput_{TCP} \leq \frac{0.7 * MSS}{RTT * packetloss^2} \tag{6.14}$$

$$MSS = MTU - header_{IP} - header_{TCP} \tag{6.15}$$

According to this model the increasing of the packet size to 9000 byte would result in a throughput 6 times bigger than with 1500 byte packets. For a virtual network jumbo frames would be as beneficial as for gigabit ethernet, because it provides a simple and scalable way to improve the network performance. There is also no bigger delay for timing critical applications, because they can still use smaller packets. Jumbo frames from other applications are also no disturbance as a jumbo frame on gigabit ethernet delays other packets the same time as 900 byte packets on fast ethernet.

Jumbo frames intrinsically reduce the IRQ rate, which is one of the limiting factors for virtual networks. This optimization is beneficial for the sending and the receiving side in contrast to TSO. For physical NIC's all intermediate nodes of a connection must support jumbo frames, witch is a crucial constraint when the Internet is used. But virtual networks could be treat as isolated networks, where the former constraint could be neglected. For communication through physical networks jumbo frames have to be translated to frames the physical NIC is capable to transmit.

For the sake of simplicity the choice of jumbo frames could be a good option for virtual networks.

6.3.5 Switched Network

In the current design, Sigma0 unnecessarily copies the packet to all VMs but the sender. For any number of VMs higher than two, there is an overhead, which could be resolved by implementing a switching strategy instead of broadcasting the packet. The two options are either to pre-configure the MAC addresses of all NICs and register them at the Sigma0 network queue, or to implement a learning switch. The former option has the benefit that it could be used as additional security feature, because only the destination VM receives the packet and the other ones cannot intercept it. Then again the second option allows to change the NIC's MAC address and replicates therefore the hardware behavior. To measure the performance data, I used a network of only two VMs, in which case there is no difference between a virtual switch and a virtual hub.

6.3.6 Zero-Copy Page Remapping

The overhead per packet can be reduced with the approaches suggested in the previous sections, but there is still the copy overhead for the DMA operations. These copies can be avoided with page remapping as described in Menon *et al.* (2006).

When the network driver in the VM registers a buffer for DMA at the NIC, the VMM determines the page host-virtual address of the guest-physical address. The VMM has to ensure that the buffer is allocated at a separate page. When the network driver triggers the sending of the network package, the VMM inspects the package to detect the destinations MAC address. With this MAC address it can instruct the virtual network switch to remap the page holding the buffer to the receiving VM and notifies it. The receiving VM then maps the host-virtual address to the receive buffer address that the NIC driver had registered.

The remapping preserves the VM isolation as the page is mapped only to one VM at one time. The transmitted packet size should be greater then 1500 byte to minimize the overhead for the mapping scheme.

6.3.7 An Efficient and Simple Model for Virtual Network

The theoretical limit for data transfer in virtual networks (6.3) is dependent from the overhead per transferred data volume (see (6.16) and (6.17)). Therefore the maximal bandwidth achievable is that of a zero copy network. There are three requirements for a zero-copy network in a full virtualization environment:

1. Data buffers have to be transmitted from the sending VM to the network model with DMA and TSO
2. DMA buffers are intercepted before segmentation and mapped to the receiving network device model
3. the receiving network model implements large receive offload (LRO) and the buffer is mapped to the output buffer of that NIC

LRO is for example supported by the Chelsio T3 10 Gigabit Ethernet adapter Chelsio (2007), but the description of LRO is out of this report's scope.

$$C_{in}^{zero} = \frac{data}{buffersize}(\#VmExit + IRQ) \tag{6.16}$$

$$C_{out}^{zero} = \frac{data}{buffersize} * IRQ \tag{6.17}$$

As there is no maximal packet size for a network that directly transmits buffers, the overhead per packet can be neglected and the limiting factor for the packet transmission becomes how fast the remap operation can take place.

Using jumbo frames in combination with DMA is a much simpler approach and could serve as a good intermediate solution. Compared to the IRQ coalescing approach there is no latency trade-off as discussed in the Section 6.3.2. The performance of this approach can still be optimized by reducing the number of copies for the jumbo frames. One copy operation can be saved by remapping pages between the different VM instances

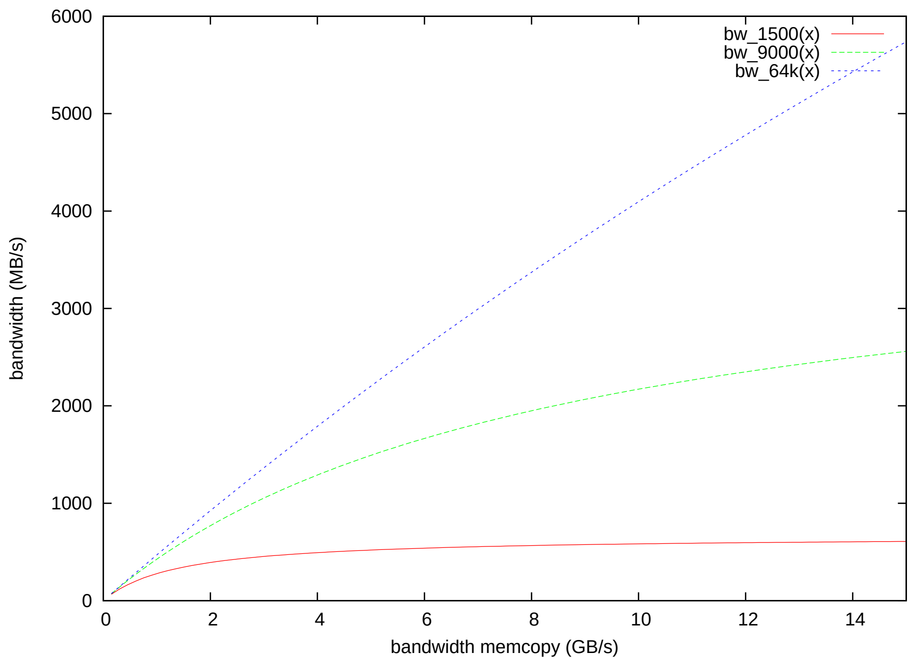

Figure 6.1: Bandwidth for frame sizes of 1500 byte, 9000 byte and 64 kbyte

as described in the Section 6.3.6. The overhead, per packet can be improved further by using super jumbo frames. Super jumbo frames are only limited by the 16bit IPv4 packet_length field to 64Kbit - 1. The following formulas show the evaluation of jumbo frames in the theoretical network model, with 3 copies per jumbo frame for the Core i7 architecture with 12.6 GB/s memory bandwidth, a CPU frequency of 2.67 GHz and 2 VMs.

$$costs_{tr} = 2 * memcpy(9000byte) + \#VmExit + 2 * IRQ + IPC + sem\_up \tag{6.18}$$

$$costs_{tr} = 2 * \frac{9000\,byte * 2.67\,GHz}{12.6\,\mathrm{GB/s}} + 1000 + 4000 + 500 + 250 \tag{6.19}$$

$$costs_{tr} \approx 9573\,cycles \tag{6.20}$$

$$bandwidth = \frac{byte * frequency}{costs_{tr}} \approx 2394\,\mathrm{MB/s} \approx 18.70\,\mathrm{Gbit/s} \tag{6.21}$$

If the frame size for external communication is restricted to the IEEE 802.3 standard, jumbo frames can still be used for the virtual network internal communication through a second virtual NIC.

7 Conclusions

In the present report it should be analyzed, whether Qemu device models can be generically ported to the Vancouver VMM. This should be examined at the example of the NE2000 network interface controller.

The starting point of my analysis was to examine both interfaces: the Qemu device model interface (see Section 2.3.2) as well as the Vancouver device model interface (see Section 2.5.1). Resulting of this analysis, I decided to join both interfaces with a generic adapter layer. The QDevice adapter made it possible to use the NE2000 NIC in the Vancouver VMM and allowed for testing the network performance on a virtualized Linux. The results of these performance measures and their interpretation are described in Section 5.2 Performance.

The QDevice adaption ayer follows the same concept of adapting code from a monolithic architecture to a component based as the DDE-Kit (Helmuth, 2001). With the concept of generic adaption it is possible to reuse unmodified sources and therefore to provide maintainability. While the DDE-Kit adapts device drivers from the Linux architecture to L4, QDevice adapts device models from Qemu to Vancouver. DDE-Kit utilizes three layers of specialization to adapt functionality at the most generic level, while the QDevice adapter uses at time of writing two layers, but it can easily be extended to more layers.

The presented adaption layer indicates that Qemu IO device models can be ported in a generic fashion. It leaves the Qemu devices unmodified and provides a clear structured interface to Vancouver. Furthermore, it bridges the gap between the component-based object-oriented VMM Vancouver and the monolithic Qemu design. A generic adapter is possible due to the well-structured components in Qemu. The implemented adapter proves the implementation costs to be quite low in code size. This indicates that Qemu device models are universally reusable in other architectures.

The measured performance of the ported NE2000 NIC is not acceptable for real-world usage. But as discussed in Section 6.3 Optimal Network Design and in Molnar (2007), the performance issues are not a consequence of the adapter-design, but inherent to the low-cost NE2000 hardware design. Nevertheless, this inappropriate performance is not only a drawback because the NE2000 NIC also serves as a compatibility option for legacy OS like DOS, Novel Netware or other OS, where no modern network drivers are available.

Another crucial result of the discussion is: the higher abstracted the hardware interface is, the higher the emulation performance in a VMM will be.

The implemented adapter keeps the device models maintainable, because it uses the unmodified sources from the Qemu project. In that way, any improvements to these could also be included into Vancouver.

This adapter shows the feasibility of a generic adaption-layer at the specific example of the NE2000 NIC. Following work can prove this statement by porting all devices required by Vancouver. By porting more devices, it could be shown that the additional implementation costs are significantly lower than the initial adapter design.

Additionally, I discussed in Section 6 several hardware features of modern network devices under the aspect of their aptitude for full virtualization. Results of this discussion are that the optimal solution would be to copy data buffers directly from one VM to another, without the conversion to network packets. But this would require the implementation of a zero-copy mechanism, which is assumed to be quite labor-intensive. Therefore, as an much simpler intermediate solution, I would rather propose a network interface supporting jumbo frames or even super-jumbo frames. With the introduced theoretical model I could show, that such a network interface would provide enough bandwidth to allow for gigabit-ethernet, even with the currently implemented copy mechanism.

Bibliography

BARHAM, PAUL, DRAGOVIC, BORIS, FRASER, KEIR, HAND, STEVEN, H, STEVEN, HARRIS, TIM, HO, ALEX, NEUGEBAUER, ROLF, PRATT, IAN, & WARFIELD, ANDREW. 2003. Xen and the Art of Virtualization.

CHELSIO. 2007 (Mar.). *cxgb - Chelsio T3 10 Gigabit Ethernet adapter driver*. http://www.freebsd.org/cgi/man.cgi?cxgb.

CLAFFY, K., MILLER, G., & THOMPSON, K. 1998. The nature of the beast: Recent traffic measurements from an Internet backbone. *In: Proceedings of INET'98*, vol. 3.

DYKSTRA, PHIL. 1999. Gigabit ethernet jumbo frames. *White Paper, WareOnEarth Communications*.

HELMUTH, CHRISTIAN. 2001 (July). *Generische Portierung von Linux-Gerätetreibern auf die Drops-Architektur*.

INTEL. 2007 (Apr.). *Interrupt Moderation Using Intel® Gigabit Ethernet Controllers Application Note (AP-450)*. http://www.intel.com/design/network/applnots/ap450.htm.

MATHIS, MATTHEW, SEMKE, JEFFREY, MAHDAVI, JAMSHID, & OTT, TEUNIS. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, **27**(3), 67–82.

MENON, A., COX, A. L, & ZWAENEPOEL, W. 2006. Optimizing network virtualization in Xen. *Page 15–28 of: Proc. USENIX Annual Technical Conference (USENIX 2006)*.

MICROSOFT. 2005 (May). *Virtual PC vs. Virtual Server: Comparing Features and Uses*. http://download.microsoft.com/download/1/4/d/14d17804-1659-435d-bc11-657a6da308c0/VSvsVPC.doc.

MOLNAR, INGO. 2007. *[kvm-devel] [announce] KVM/NET, paravirtual network device*. http://www.mail-archive.com/kvm-devel@lists.sourceforge.net/msg00824.html.

NATIONAL SEMICONDUCTOR. 1996. *DP8390D - NIC Network Interface Controller*. http://www.national.com/opf/DP/DP8390D.html.

POPEK, GERALD J., & GOLDBERG, ROBERT P. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, **17**(7), 412–421.

ROTHENBERG, JEFF. 1999. *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation*. Council on Library & Information Resources.

STEINBERG, UDO, & KAUER, BERNHARD. 2008. *NOVA OS Virtualization Architecture*. Tech. rept.

TUCKER, S. G. 1965. Emulation of large systems. *Commun. ACM*, **8**(12), 753–761.

WALDSPURGER, CARL A. 2002. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, **36**, 181–194.

ZEC, M., MIKUC, M., & ZAGAR, M. 2002. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. *In: Proceedings of the 10th SoftCOM*, vol. 2002.