LEHRSTUHL FÜR BETRIEBSSYSTEME

Migration zur Laufzeit von virtuellen Maschinen zwischen heterogenen Hostsystemen

Live Migration of Virtual Machines between Heterogeneous Host Systems

Jacek Galowicz
Matrikelnummer: 285548

Masterarbeit
an der
Rheinisch-Westfälischen Technischen Hochschule Aachen
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Betriebssysteme

RWTH AACHEN UNIVERSITY

Betreuer:	Dr. rer. nat. Stefan Lankes
	Dipl.-Inf. Udo Steinberg (*)

(*) Intel Labs Braunschweig

Kurzfassung

Der NOVA Microhypervisor und der mit ihm in Kombination verwendete Userlevel Virtual Machine Monitor schlagen die Brücke zwischen hocheffizienter Virtualisierung und Mikrokern-Betriebssystemen. Virtuelle Maschinen stellen in NOVA einen Weg dar, veraltete Software wiederzuverwenden und allgemein Betriebssysteme von der darunterliegenden Plattform logisch zu entkoppeln. Live-Migration ist eine Technik, mit der sich virtuelle Maschinen zwischen zwei Host-Systemen zur Laufzeit transferieren lassen, ohne diese dabei für einen längeren Zeitraum unterbrechen zu müssen. Das Ziel dieser Arbeit ist es, neue Möglichkeiten zu finden, den VMM insofern zu erweitern, dass virtuelle Maschinen zwischen heterogenen Host-Systemen migriert werden können. Für das Beispiel von Netzwerkschnittstellen wird ein Mechanismus entworfen, der mit Hilfe von ACPI Hotplug-Features die Live-Migration virtueller Maschinen ermöglicht, die schnelle pass-through-Netzwerkkarten zur Kommunikation benutzen. Um stehende TCP-Verbindungen migrationsübergreifend intakt zu halten, wird das bestehende Netzwerkkarten-Modell um die Fähigkeit erweitert, seinen eigenen physischen Umzug im Netzwerk an umliegende Netzwerkteilnehmer zu propagieren. Sämtliche neu implementierten Mechanismen kommen ohne Änderungen am Sourcecode des Gastsystems aus. Eine mögliche Gastsystem-Konfiguration wird präsentiert, die mit dynamisch entfernten und wiederhinzugefügten Netzwerkkarten umgehen kann, während sie Dienstanfragen über Netzwerk bedient. Die vorgestellte Lösung ist völlig unabhängig von den auf den verschiedenen Hosts eingesetzten Netzwerkkarten-Modellen. Am Ende dieser Arbeit wird die Leistung des Prototyps ausgewertet und es werden zukünftige Einsatzmöglichkeiten beschrieben, sowie Optimierungspotenzial in der bisherigen VMM-Architektur in Hinsicht auf Live-Migration angemerkt.

Stichwörter: LfBS, Master-Arbeit, Microkernel, NOVA Microhypervisor, Virtualisierung, Live-Migration, Pass-Through von PCI-Peripheriegeräten, ACPI, Hot Plugging, GNU/Linux, NIC Bonding Treiber, gratuitous ARP

Abstract

The NOVA microhypervisor and its companion user-level virtual-machine monitor facilitate the construction of systems with minimal application-specific trusted computing bases. On NOVA, virtual machines provide a way for reusing existing legacy software and for decoupling guest operating systems from the underlying platform hardware. Live migration is a technique to transfer a virtual machine from one host system to another with minimal disruption to the execution of the VM. The goal of this thesis is to explore how the existing live migration feature of the VMM can be extended, such that virtual machines can be moved between heterogeneous host systems with different CPU capabilities and different host devices. Using network devices as an example, a mechanism is designed to migrate virtual machines equipped with fast pass-through network interfaces, utilizing native ACPI hot plugging mechanisms. To preserve standing guest TCP connections over migrations, the network interface model is extended to make the host propagate the physical movement of the guest within the network to other network participants. Guest system source code is left untouched. It is shown how to configure guest systems to enable them for dealing with disappearing/reappearing network interfaces while servicing network requests during the migration of themselves. The presented solution is device type agnostic and can also deal with devices which differ from host to host. The prototype implementation is evaluated, future possibilities are outlined and possible performance optimizations are described.

Keywords: Chair for Operating Systems, Master Thesis, Microkernel, NOVA Microhypervisor, Virtualization, Live Migration, PCI Device Pass-Through, ACPI, Hot Plugging, GNU/Linux, NIC Bonding Driver, Gratuitous ARP

Acknowledgements

I would like to express my deep gratitude to Stefan Lankes and Michael Konow, for enabling my internship and subsequently my master thesis at Intel Labs Braunschweig. Also, I would like to thank Udo Steinberg, my always helpful and patient supervisor, as well as Julian Stecklina and Bernhard Kauer from the operating systems group of TU Dresden for their valuable advice. Finally, special thanks goes to all the colleagues and new friends for making my year in Braunschweig very worthwhile.

Contents

List of Figures

1. Introduction

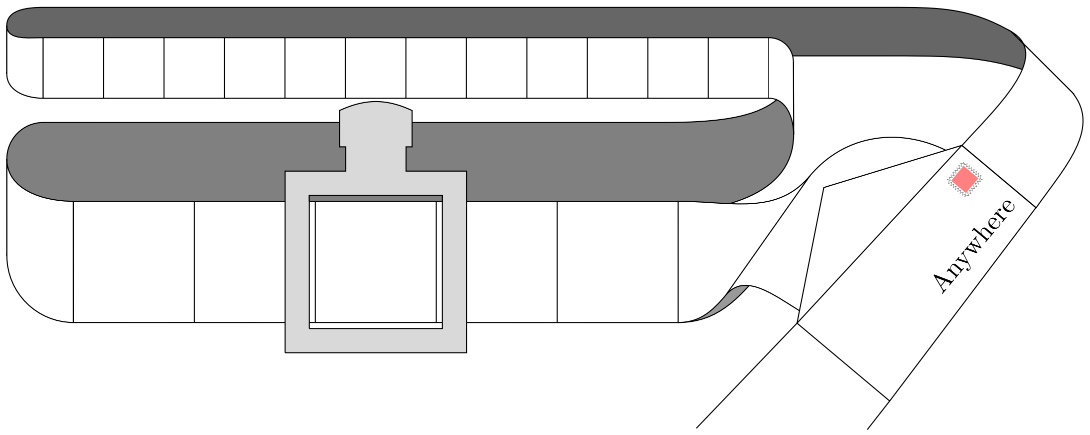

Figure 1.1.: Live Migration from a Very High Level Perspective

The human ability to abstract can take on an interesting scale. This becomes especially apparent when looking at current computer technology. Humans do, for example, write and run computer programs which can access *data files* with the impression of unlimited size. At the same time an operating system transparently abstracts the cylinders, heads and sectors of hard disks or other media to provide the promised plain and linear view onto this file's data space. The program being run is also provided with the illusion of having access to the whole memory space of the underlying computer system. Then again, the operating system, utilizing hardware mechanisms for this, transparently translates access to mere *virtual* memory addresses to real *physical* ones. One might expect that the abstraction is overcome when inspecting the operating system as it represents the lowest layer of software executed by the computer system. But with increasing regularity, even *the computer system itself* is an illusion. A *hypervisor* system running at an even lower software level multiplexes the hardware to be able to run multiple *guest* operating systems while logically isolating each other.

All this effort to delude software with abstract views of the system by putting complex layers of software beneath it has the purpose of providing an abstract runtime environment. Applications created for such an abstract runtime environment are capable of running with minimal knowledge about the underlying system components. Only with such a system of stacked layers of abstraction it is possible to write and run portable software, being safely isolated from other code and data at the same time.

Implementing a layer of software which provides some kind of delusion like infinite files, linear memory spaces or whole computer systems is also called *virtualization*. The area of virtualization which underwent most change and progress in the last decades is the virtualization of whole computer systems. Guest systems are run in so-called *virtual machines* (VM). As technological progress makes it increasingly efficient to run software within VMs, virtualization solutions are becoming attractive in more and more areas of computation. When virtualizing classic *server* systems, it is possible to consolidate multiple systems on one physical host. Assuming that servers usually do not use their hardware to capacity all the time, it is possible to reduce energy and space consumption this way. VMs can also be used to run untrusted or legacy software.

Another famous advancement which emerged from virtualization technology in the last years is *live migration*. VMs can be regarded as mere memory states on the executing host system. Therefore they represent plain data which can be sent anywhere over network. A logical consequence is the idea to transfer VMs with the aim to execute them on other hosts. Virtualization layers which implement this idea *decouple* software from the hardware it is executed on. Now it is not only possible to consolidate multiple VMs on single host machines, but also to rebalance them over hosts during runtime. The enormous amount of flexibility gained from this feature is an important contributory cause for the ongoing pervasiveness of virtualization in industry.

However, in some areas virtualization is stretched to its limits. Where high I/O performance like in computer networking with maximum throughput is needed, virtualization becomes a bottle neck. As it is still not possible to virtualize e. g. 10 Gbit/s network interfaces, it came into fashion to give VMs full access to individual host devices. Current hardware supports to do this in a strictly isolated way. Unfortunately, this prevents live migration, since such a VM is not representable by a plain memory state any longer. A dramatic consequence of this constraint is that users are forced to decide between the flexibility of virtualization *or* I/O performance.

The focus of this thesis is to explore possibilities to provide a solution for this problem. On the example of VMs employing fast network interfaces of their host system, it is shown how to make them migratable again. Beginning with an overview of involved technologies for this project, the second chapter clarifies all terms used throughout the thesis. A precise definition of the project goal is given in chapter three. After discussing similar solutions other research teams came up with to solve the same problem, design decisions are made and justified. Needed preliminary work and new subsystems are identified and high-level descriptions are given. Chapter four describes the actual implementation in detail. A performance evaluation of the implementation follows in chapter five. Finally, chapter six gives a roundup of the project achievements and gives further suggestions regarding software architecture optimizations for deploying performant live migration. An outlook section sums up some philosophical impulses about what potential lies in future platforms which are designed with the new feature in mind.

2. Background and Related Work

This chapter explains all subsets of virtualization technology needed to describe and solve the problem presented in this thesis. Being the foundation of all following subsystems, virtualization in general is defined and outlined in different facettes. An important part is hardware support which elevated virtualization technology to the maturity enabling its dramatic growth in the IT industry. The Intel VT hardware extensions are described as a prominent example. Marking a yet unsolved bottleneck in virtualization, massive I/O with virtual peripheral devices like network cards does not lead to the same effective data rates as real hardware in bare metal machines provides. The IT industry usually works around this by giving virtual machines isolated access to real hardware devices. Although this is an elegant solution in many cases, the utilization of real hardware in virtual machines prevents the use of live migration. A short analysis of this bottleneck is followed by explanations about the common workaround of real device pass-through. Live migration is a very complex topic in research and also a key feature to this project, thus explained in another section. The last section contains an explanation of how PCI hot plugging is done in general and in the example of ACPI hot plugging.

2.1. Virtualization

> *Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications.*
>
> Robert P. Goldberg, 1974

In the early days of computer technology, mainframe computers provided virtualization to multiplex one big physical machine among multiple users. This was done for cost reasons. Nowadays, costs are no longer an issue for buying computer systems. Nevertheless, virtualization has a big revival in the IT industry because of its variety of advantages. These will be clear after defining the term *virtualization* and explaining the technical background:

Definition (Computer Virtualization)**.** *Virtualizing a computer means creating virtual instances of its hardware which do not physically exist.*

An application on a virtual computer system must run with the identical effect to executing it on a physically existing computer system. The physical system providing the virtualization mechanism is called *host machine*, while the virtual computer system is a *virtual machine* (VM) or *guest machine*. Being mostly implemented in software, the virtualization mechanism itself is usually embodied by a program called the *virtual machine monitor* (VMM), or *hypervisor*.

In his PhD thesis, Robert P. Goldberg dinstinguished between two fundamental types of hypervisors [10]:

Type I Bare metal hypervisor (Examples: Xen, KVM, lguest, NOVA)

Type II Extended host hypervisor application running on the host OS (Examples: QEMU, Bochs)

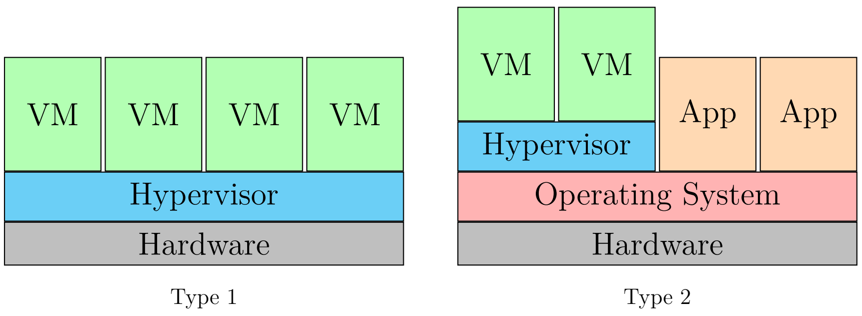

Figure 2.1.: Layer Diagrams of the Different Hypervisor Types

Type I hypervisors are directly included into the OS kernel. This can be any commodity OS like Linux with a special virtualization module like KVM [20]. Alternatively, the OS can be primarily designed for virtualization, like Xen [2]. The hardware is then only accessible by the user via virtual machines. This type of virtualization provides best performance, because it is done with highest privilege level and low overhead, possibly even using hardware support for accelerated virtualization.

Type II hypervisors can usually be executed in user space without any special privileges. While they are less performant because of their lacking use of hardware support, they can implement totally different architectures, which renders them more flexible. A very impressive example for this type is Fabrice Bellard's Javascript x86 emulator[1] which boots a small Linux system within an internet browser window in seconds. As the underlying project of this thesis uses a type I hypervisor, type II hypervisors will not be minded in the following.

As of today, virtualization is often used to host many VMs on few physical machines. This can reduce power consumption, since many systems just idle in the majority of time. In data centers it can also reduce the need for big cooling plants and floor space. Another interesting use case is the isolation of untrusted software

[1]`http://bellard.org/jslinux/`

in VMs instead of running it together with trusted software using sensitive data on the same platform. As computer systems evolve, it can also be useful to be able to run VMs emulating legacy hardware for needed legacy applications.

2.1.1. Faithful Virtualization Versus Paravirtualization

In general, virtualization usually means *full* virtualization or *faithful* virtualization. Virtualizing a computer then means that every memory and I/O read/write has the same effect as on existing hardware. The hardware architecture visible to the software is then completely identical to a possible configuration of real hardware. This has the obvious advantage that software being run on such a VM does not need to be rewritten in any regard. On the downside, faithful virtualization has several implications regarding performance. Any use of devices outside the CPU involves I/O communication. Reads and writes from or to I/O ports as well as most memory mapped I/O (MMIO) registers have to be trapped by the hypervisor. Trapping the VM means switching back from guest to host (VM exit), processing the I/O access to change the state of the VM accordingly and switching back to the guest again (VM resume). A switch between host and guest in both directions is in general called a *VM transition*. This approach is called *trap and emulate* and represents the main bottleneck when virtualizing hardware devices.

One common concept to improve this, is reducing the number of VM exits. If it is possible to exchange more information with the hypervisor on every VM transition while at the same time reducing transitions, device virtualization would be more efficient. The first project tackling this idea was the *Denali isolation kernel* [35]. Whitaker et al. loosened the constraint of virtualizing the exact host architecture to allow simplifications in the virtualized hardware interface. Instead of expressing communication with the hardware as sets of I/O accesses, a paravirtualized guest system communicates with the hypervisor using *hypercalls*.

The underlying virtualization solution does not use paravirtualization, which introduces a certain amount of complexity to the presented problem.

2.1.2. Intel VT Virtualization Hardware Extensions

Several Intel CPUs today provide the Intel *Virtualization Technology* extensions (VT) [14]. Formerly codenamed as *Vanderpool*, they are now available in two versions: VT-x for x86 processors and VT-i for Itanium processors. These extensions provide two modes to operate the CPU in: *VMX root operation* and *VMX non-root operation*. In general, the hypervisor will operate in the former mode, while all VMs will only operate in the latter. AMD processors provide a similar architecture for hardware assisted virtualization called *Secure Virtual Machine Architecture* (SVM) [1].

VMX root operation is very similar to non-VMX operation, although it provides the hypervisor software the possibility to configure the behavior of VMX non-root operation for VMs. VMX non-root operation in turn, is restricted, which is not detectable by VMs. Whenever a VM executes restricted instructions or does I/O, a VM exit is initiated. This mechanism is active in any privilege ring, allowing guest software to run in the privilege rings it was initially designed for. It is then the duty of the hypervisor to maintain the state of the virtual machine in a way a real machine would react. This can be done to such an extent that the VM is unable to distinguish its environment from real hardware.

The previously mentioned possibility to configure the behavior of the processor in VMX non-root mode is given via the 4 KB large *virtual machine control structure* (VMCS). A hypervisor can maintain one VMCS per VM, or in case of multicore VMs, one VMCS per virtual processor. Every processor in the host system has a VMCS pointer which can be set to the VMCS of the next VM to be executed.

A very important subsystem of this hardware virtualization support are *extended page tables* (EPT) [4]. Usually, the memory space processes use is merely virtual and translated by the *Memory Management Unit* (MMU) to real physical addresses as they exist in the memory hardware. Guest operating systems also use this feature, but the memory range they are provided with is not the *real* physical memory range as it would be in a real machine. Thus, the translation to guest-physical addresses by the guest page tables corresponds to host-virtual addresses. VMs running on non-VT hardware have to be trapped by the VMM whenever the guest OS tries to load or switch its own page tables. The trap handler of the VMM would then complete the guest-virtual to guest-physical mapping proposed by the guest to a host-physical mapping and store it in a *shadow page table*. Then, the shadow page table would be the one actually be used by the MMU. This layer of indirection was significantly simplified with the introduction of EPTs. Guest operating systems can now natively maintain their page tables. On every page fault, the MMU traverses the guest page tables and then the nested page tables to provide the corresponding mapping without any costly VM exits. [30, 9]

2.1.3. Challenges in Virtualizing Fast Network Adapters

While it is possible to virtualize processor and memory components of a VM with near native performance, I/O virtualization is still a major cause for performance degradation. Each interaction between the guest OS and any of its devices involves I/O, hence needs to undergo expensive VM transitions and device emulation steps. The device model itself might also involve multiple layers of software for isolating and multiplexing between several VMs. Especially high-throughput network interface devices involve very frequent I/O interaction with the OS to be able to handle the very high rates of incoming packets.

To get some understanding about the actual timing dimensions needed to drive a 10G NIC to full speed, see Figure 2.2. Network packets can have netto sizes between 64 B and 1518 B. Before a packet is transmitted over copper cable, an 8 B preamble as well as about 12 B interframe gap is appended to the packet by the NIC. If the receiving system exceeds a certain amount of processing time, it effectively throttles the packet throughput.

$$\begin{aligned}
n_1 &= 64\ \frac{B}{Packet} + 8B\ preamble + 12B\ IFG = 672\ \frac{bit}{Packet} \\
n_2 &= 1518\ \frac{B}{Packet} + 8B + 12B = 12304\ \frac{bit}{Packet}
\end{aligned}$$

$$\begin{aligned}
t_1 &= 672\ \frac{bit}{Packet} \Big/ 10 \frac{GBit}{s} = 67.2\ \frac{ns}{Packet} \\
t_2 &= 12304\ \frac{bit}{Packet} \Big/ 10 \frac{GBit}{s} = 1230.4\ \frac{ns}{Packet}
\end{aligned}$$

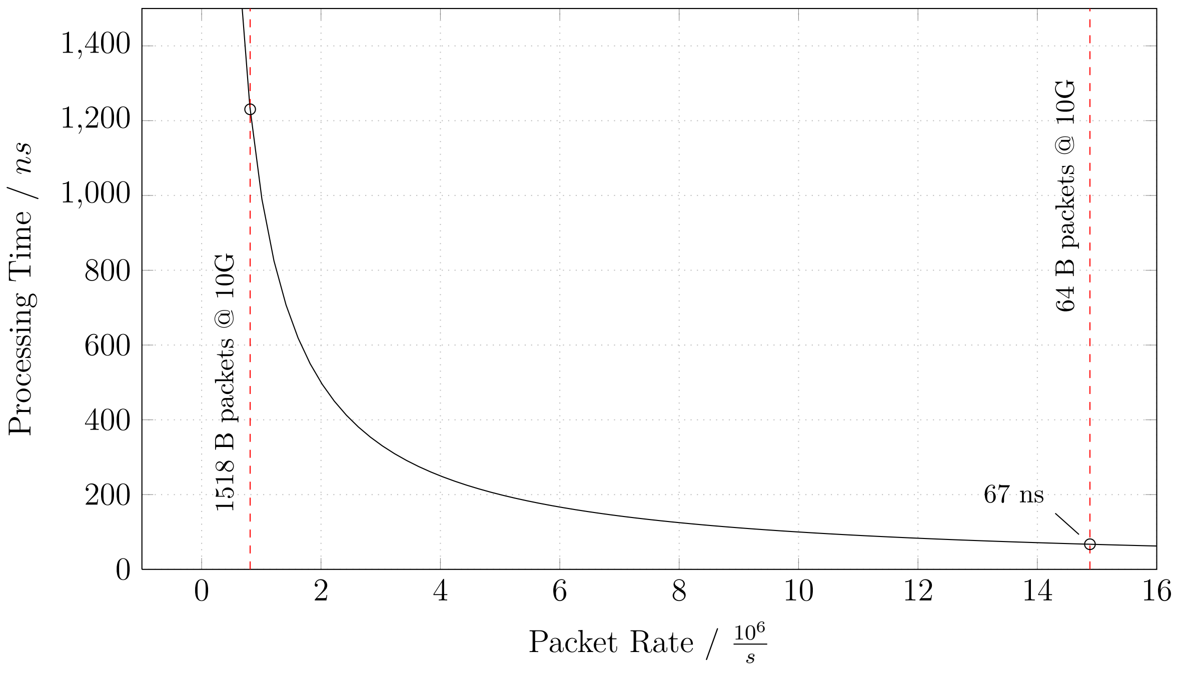

Figure 2.2.: Packet Processing Time with Varying Packet Sizes

Processing times of down to 67 ns have to be achieved to enable for full NIC performance in VMs. Even with current hardware assistance mechanisms for virtualization it has not yet been shown how to meet this timing constraint.

A common approach to tackle this problem is paravirtualization, which is already deployed and broadly used in the IT industry. Virtio is a very prominent example for simplified device interfaces in paravirtualization [31]. With heavily simplified device models and drivers, it is possible to dramatically shrink emulation complexity and frequency of I/O communication between guest OS and its virtual devices. A downside of this approach is the inherent virtualization-awareness, leading to guest OS code change, and the remaining high amount of CPU overhead compared to native environments. There have been efforts to optimize the path any network packet has to take through the software stack to optimize the throughput for paravirtualized devices. However, this still did not reach native performance as it does not handle the I/O problem itself [7, 27].

2.2. Real Device Pass-Through

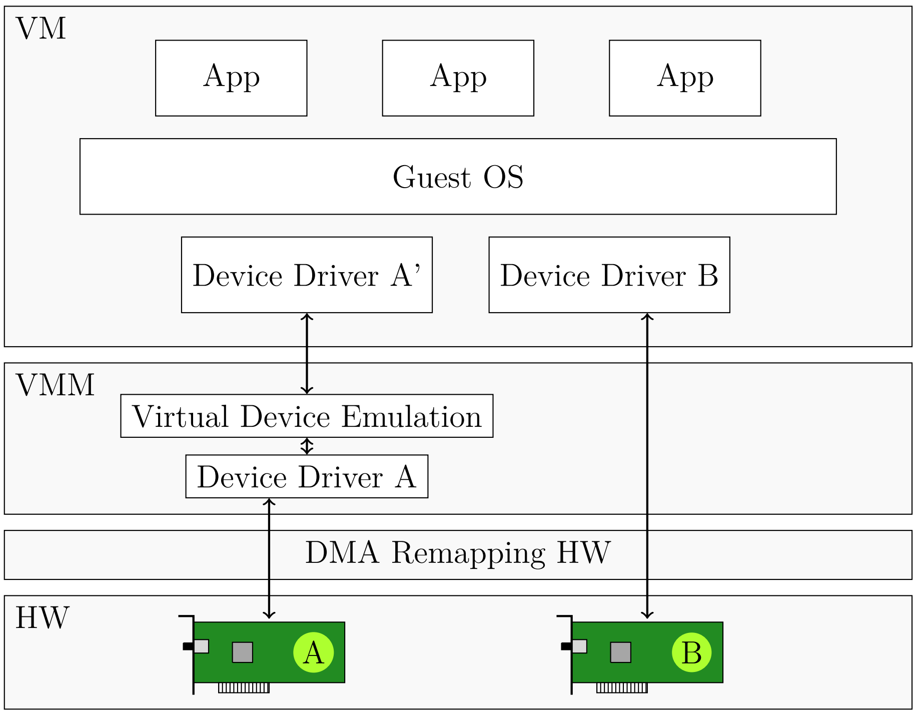

Figure 2.3.: A VM Using a Virtual Device and a Physical Pass-Through Device

Another approach to provide VMs with devices with high throughput-rates is passing the VM control over actual physical devices. A completely virtual computer system would then work with single isolated physical devices belonging to the host system. Figure 2.3 illustrates this configuration. The VM can see two devices, of which device A' is a virtual model of physical device A and device B is directly passed through. Passing through devices to a guest OS like this enables for near native efficiency. The Intel VT-d extensions complement the existing VT extensions for this purpose. The efficiency is still only *near* native because the interrupt and DMA emulation implies a certain amount of overhead. Nevertheless, this approach is a widely accepted solution in the IT industry.

2.2.1. IOMMU

To give a VM access to a physical device in an efficient way, it is necessary to implement a number of mechanisms. I/O ports need to be accessible from the guest, as well as memory mapped I/O page access has to be translated accordingly. Interrupts, which are the common way to enable devices to notify the OS about hardware events, need to be routed to the appropriate VM. Direct Memory Access (DMA) initiated by physical devices, needs to be narrowed down to pages mapped to the respective VM to enhance security and stability of both host system and other VMs. The hardware device capable of handling this kind of device I/O is called *I/O Memory Management Unit* (IOMMU) [3].

Intel's VT-d extensions are represented by a generalized IOMMU implementation located in the north bridge of the system, as shown in Figure 2.4. The I/O ports as well as memory pages and interrupt routes of a system can be partitioned into *protection domains*. These protection domains are transparent to any VM [15].

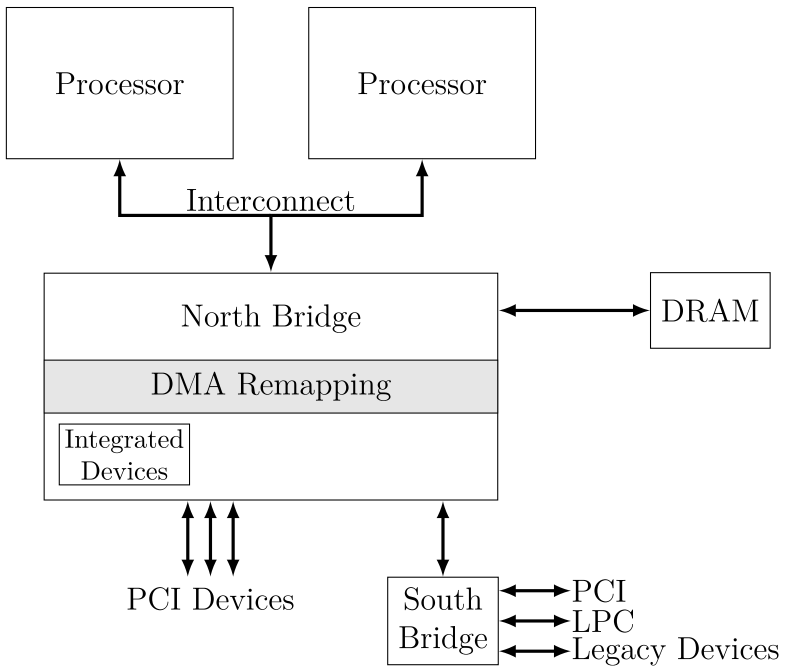

Figure 2.4.: DMA Remapping of the IOMMU in the Platform Topology

2.2.2. PCI Bus

The *Peripheral Component Interconnect* (PCI) bus was invented to connect peripheral devices within computer systems. It found widespread use in mobile, desktop, workstation, server, and embedded computing after its first implementation in IBM PC compatible systems. Traditionally, the PCI bus was connected to the CPU bus via the *north bridge* controller on the motherboard. Using PCI, it is possible to communicate with extension hardware via programmed and memory mapped I/O. PCI devices share four interrupt lines on the bus. These can emit CPU interrupts which are directly routed either to interrupt controller or APIC bus. Newer PCI revisions introduced the *Message-Signaled Interrupt* (MSI), which is basically a memory write access translated into an interrupt cycle by the north bridge.

The PCI bus[2] was designed to represent an interface for plug-in cards, but in practice most computer motherboards also already contain onboard PCI devices. Generally, a *root bus* exists to which up to 32 PCI devices can be connected. To add even more devices, a *PCI bus bridge* can be connected instead of a device. Such a bus does again support the same number of PCI devices. Before a device is properly configured to be reachable via programmed or memory mapped I/O reads/writes, it has to be addressed by its *Bus, Device, Function* (BDF) identifier. These three numbers identify the bus, device number slot and device function, if the PCI device embodies multiple devices in one.

2.3. Live Migration of Virtual Machines

In virtually every branch of computing it is desirable to be able to move executing software from one machine to another without the need of a restart. Having such a mechanism, it is possible to take a computer down for maintenance without having to stop any important program running on it. Another use case is the rebalancing of the CPU loads between multiple running servers. It would also be advantageous to move the processes of relatively idle hosts together to a smaller set of hosts to be able to shut the majority of machines down in order to save energy. Especially in high performance computing it would reduce latency and maximize throughput between compute nodes, if they were always in minimal reach, which could be achieved dynamically with process migration. In heterogeneous clusters consisting of machines with different amounts of processing power, processes could be migrated to the fastest available nodes.

To successfully migrate a process, all resources it depends on on the source platform, i.e. its *residual dependencies*, have also to be provided on the destination platform as well. Migrating resources of a process, such as threads, communication channels, files, and devices is not possible in all cases. Either services like system calls, file systems, etc. are available on every node or requests are redirected to the home node being able to provide them. Regarding Figure 2.5, the process state is *cut out* of the running system along the interfaces demarked by ③ and ⑦. The residual dependencies which need to be fulfilled on the target host are now both the complete ISA as well as the part of the complete operating system state which is relevant to the running process. Especially the latter might dramatically raise the complexity of the migration environment when keeping in mind that it is certainly

[2]From hardware view PCI is not a bus any longer. However, from software view, it still is for compatibility reasons.

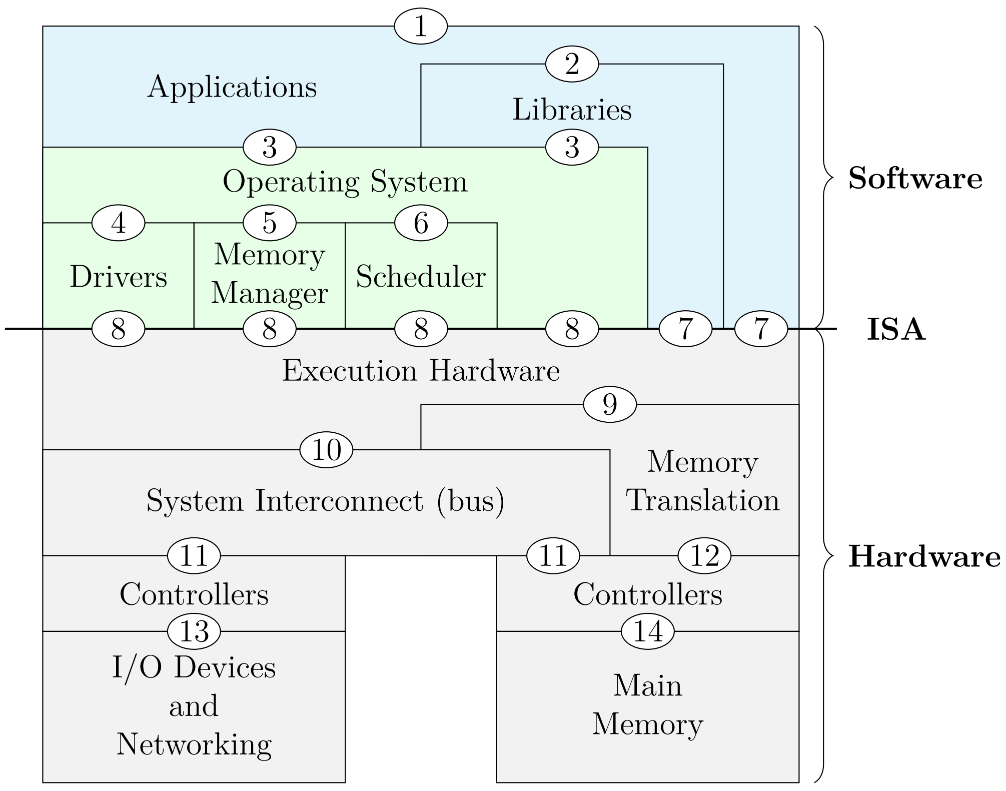

Figure 2.5.: General Computer System Architecture with Numbered Communication Interfaces [32]

impractical to maintain perfectly equal operating system versions between all hosts. The ISA is not allowed to differ between hosts at all. [28, 32]

From the point of view of a VMM, all states of guest processes and even guest operating systems are represented by the state of a plain range of memory, i.e. the guest memory. This could be written to disk, or sent over network, of course. Migrating a whole VM instead of a process would then, again regarding Figure 2.5, correspond to a cut at the interfaces ⑦ and ⑧, which separate hardware from software. Having only the ISA and the hardware state as a residual dependency is advantageous. Hardware can be virtualized efficiently and the state of virtual hardware models does usually not have residual dependencies to the layers the VMM is run on. In practice this means that a VMM with the same virtual hardware configuration as the source VMM can be started on the target host. This new VMM could then just receive the guest state to overwrite its own guest memory and guest device model states with it. [6]

After the migration of the whole VM, it will run independently on the target host. The source host can be switched off. As opposed to process migration, no remaining residual dependencies make it necessary to keep it running. This big simplification to process migration is bought with the need to transfer a whole operating system, inducing network transfer volume overhead. However, the transfer of data in the order of gigabytes is not an issue with modern 1 Gbit/s or even 10 Gbit/s network interfaces. Live migration has become a feature of virtualization with broad acceptance in industry. Chen et al. even consider it one of the most important features in this area of computing [5].

2.3.1. Cold Migration Versus Live Migration

In general, a running VM has to be stopped before its state can be written to disk or sent over the network to be resumed on another host agian. The state of a frozen VM is consistent, which is the requirement to let its execution continue at a different point in time or on a different host. This approach is also called *checkpointing*. Moving the checkpoint of a VM is a *cold migration*.

However, in many use cases it is not desirable to have to *pause* a VM to physically move it. It may be running an important service and for availability reasons it has to be moved away from a faulty host, for example. Checkpointing the VM and transfering it would then lead to unacceptable downtime. Transfering the state of a running VM is problematic because at least parts of the state which is being transfered, change at the same time. For this reason, a mechanism is needed to *track* which parts of the state changed to resend them immediately. Sending changes while the VM state continues to change, leads to a recursive chain of transfer rounds which needs to be cut at some point. In a last round the VM would be frozen to send its last state changes, leading to a minimum downtime. This approach is called *pre-copy migration* and was first demonstrated by Christopher Clark, et al [6].

Other approaches, to keep the VM state consistent during migration, exist. A direct alternative to pre-copying all guest memory pages is *post-copying* them on the target platform. Hines et al. demonstrated this in combination with a demand-paging mechanism [13]. Haikun et al. presented another innovative solution. To avoid a lot of network traffic, they demonstrated how to *trace* all VM-interaction with the outside world before the migration and *replay* this on the target machine to keep both synchronized [26].

The underlying migration algorithm this thesis is based on, implements the pre-copy approach, hence all others are ignored. Figure 2.6 shows a qualitative network traffic throughput graph of a VM during a fictional live migration. In this scenario, the live migration is initiated at second 1. The VMM sends the guest state which leads to throughput degradation of the VM since both use the host network interface. It is possible to additionally throttle the guest network connection to provide the VMM a higher throughput-rate to finish the migration earlier. Four seconds later, the state synchronization between source and destination VMM is finished and the VM is frozen. Transfering the rest of the state changes to the target VM and resuming VM execution on the destination host takes 0.5 seconds in this scenario. This downtime gap can be very critical and needs to be reduced to a minimum, depending on the use case. Game servers, multimedia streaming servers or other low-latency services, for example, are downtime-sensitive workloads. In the end,

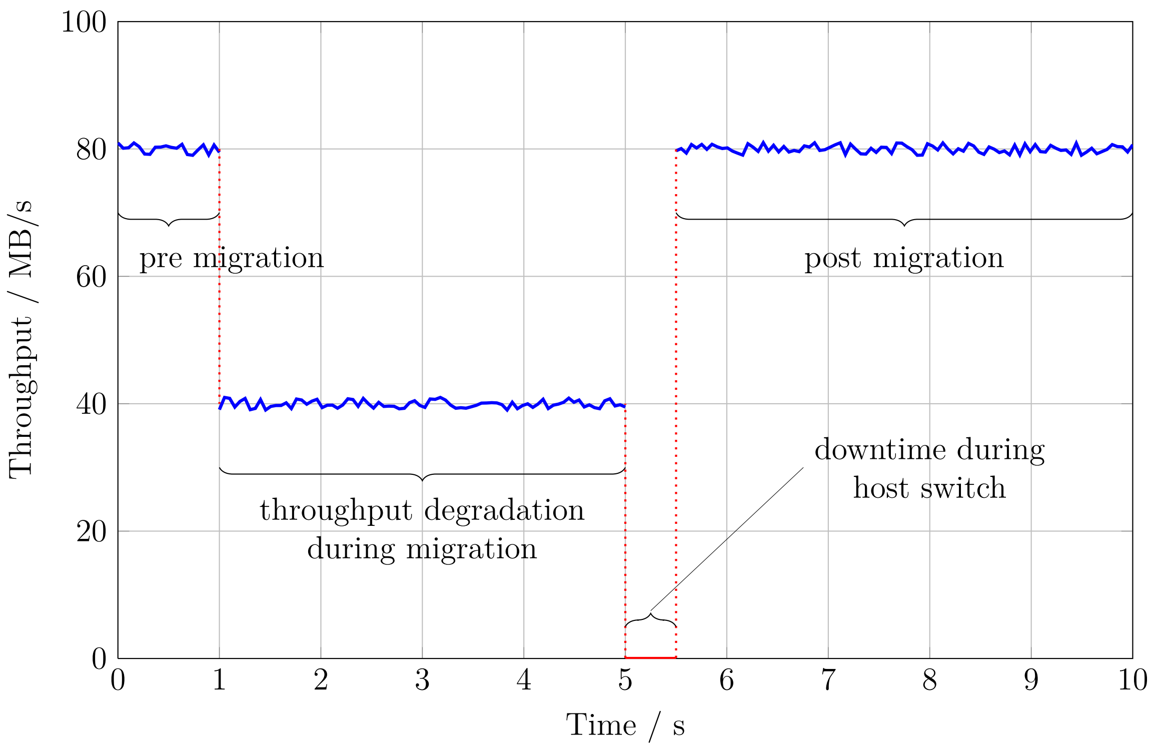

Figure 2.6.: Network Throughput of a Fictional VM Migration

the downtime gap predominantly depends on the network latency, the maximum network throughput, and the size of the *writable working set* (WWS) of the VM. The WWS is the set of memory pages which is continuously modified by the VM. As soon as the pre-copy algorithm reduces the set of pages to be transfered down to the WWS, it will need to stop the VM. With faithful virtualization, the size of the WWS can hardly be manipulated. This means that the downtime is heavily dependent on the actual workload. From second 5.5 on when the migration is completed, the guest VM returns to maximum network throughput. The network throughput on the destination host does of course depend on the maximum throughput of the network interface of the host and on the possible network load caused by other VMs.

2.3.2. Migration Friendly Hardware

In a perfect world it would be possible to read out the state of a piece of hardware and write it back to another identical device, which is the requirement for migrating a VM using it. This is not easily possible with existing hardware. The state of a device is only visible to the operating system via its registers. Hardware registers are usually different in their read/write behavior compared to a portion of memory. *Read-write* registers which have no side effects after access are migratable, however there also exist registers with side effects which break with any read-restore hopes. *Write-only*, *read-write-clear*, *write-clear*, etc. registers cannot be read and/or written back to hardware.

2.4. PCI Hot Plugging

For availability as well as extensibility reasons, it has always been important for the administration of server computers to be able to substitute parts of the hardware without shutting down the system. Pulling out a hardware device of—or putting it into—a running system is called *hot plugging*. Obviously this cannot work if the software is not involved into the process of removing or adding a device it shall employ. Some kind of controller is needed to provide a base of negotiation between user, operating system and hardware. The process of hot removing a device generally looks like the following, with the example of PCI cards:

1. The administrator determines that a PCI card must be removed and notifies the running operating system about this plan. Notifying is usually done with a shell command or the press of a button on the mainboard.
2. The operating system unloads the driver and notifies back to the hardware that it can power off the device.
3. The system unpowers the PCI slot of the device and notifies the user e. g. via a flashing LED on the mainboard that the device is now without power.
4. The administrator removes the device.

Hot adding a device into an unpowered PCI slot works analogously. Hot plugging is not only limited to PCI devices; hardware exists which is even able to deal with hot plugging of CPUs.

There are mainly three different types of PCI hot plug controllers [36]:

ACPI Hot Plug Since laptop computers supporting docking stations have a similar ACPI-mechanism providing hot plugging of additional hardware, general PCI hot plugging can be implemented with standard ACPI *general purpose events*.

SHPC (Standard Hot Plug Controller) The PCI-SIG defines a standard PCI hot plug controller [11].

Proprietary controllers Several companies provide their own proprietary hot plug controller solutions.

Although the Linux kernel as of version 2.6 supports most hot plugging standards, it is preferable to pick the one which brings the least implementation complexity to the hypervisor. A guest OS kernel already supporting hot plug events which was configured without any virtualization in mind, already provides a perfect interface exploitable by a hypervisor. This way live migration can plainly be hidden from a faithfully virtualized system behind hot plugging events.

2.4.1. ACPI Hot Plugging

In contrast to other solutions, the ACPI variant of a hot plug controller is based on a very simple and cleanly defined open interface. Thus, an ACPI controller model implementation introduces the least complexity into the hypervisor.

ACPI is an abbreviation for *Advanced Configuration and Power Interface* and stands for an open standard for device configuration and power management controlled by the operating system. The first specification was released 1996 by Intel, Microsoft, and Toshiba. [12]

Mainboards providing ACPI support, write certain tables into main memory of the system early at boot time. These tables describe all ACPI-related capabilities to the operating system. Some tables provide fixed information at fixed offsets, others describe system functionality in form of methods doing I/O communication with the hardware. The latter form of description is expressed in *ACPI Machine Language* (AML) compiled from *ACPI Source Language* (ASL), for which the operating system has to implement an interpreter. This way the operating system can assume certain interfaces with known functionality implemented by AML-procedures. Intel provides a compiler for ASL which is also able to decompile AML code [18]. See Figure 4.4 on page 38 for an example table configuration. [17]

2.5. Microkernel Operating Systems

Typical operating systems unite all hardware handling, virtual memory management, interrupt handling, scheduling, file systems, device drivers, etc. in one big piece of software binary: the *kernel*. Kernels like this are called *monolithic* kernels. Hardware architectures like x86 provide security enhancements like *privilege rings* to exclusively allow only certain processes to manipulate the whole system state. Monolithic kernels usually let applications run in unprivileged *user space* (ring 3 on x86 systems), while all kernel threads run in privileged *kernel space* (ring 0). This configuration already represents a great increase of system security and stability, because it allows to isolate applications from direct system access and also from each other. A crashing application can neither access data of other applications, nor crash the whole system.

However, in practice, software is rarely free of bugs. These can obviously occur in kernel code, which is a potential problem for the security and stability of the system. A malicious device driver, for example, can already jeopardize the integrity of the whole system. The portions of code in the system which have to be trusted, are called the *Trusted Computing Base* (TCB). To inherently improve the security and stability of a system, the minimization of its TCB is desired. Nevertheless, the kernels of monolithic operating systems steadily increase in size, to support the ever growing variety of new hardware components new systems can consist of.

An alternative to this system architecture are *microkernel* operating systems. Architects of microkernel operating systems achieve a reduction of the TCB by moving as much code as possible out of kernel space. In such a system, virtual memory handlers, file systems, interrupt handlers, and device drivers are individual user space applications, each, equipped with privileges reduced to the absolute minimum. The microkernel only implements three key abstractions: *Address spaces*, *threads* and *Inter Process Communication* (IPC). Figure 2.7 shows the typical layers of both monolithic and microkernel systems. User space applications in monolithic operating systems ask for system functions like e. g. file system access via *system calls*. In microkernel operating systems, applications communicate with the respective server applications via IPC. The great advantage of this approach is, that a crashing device driver, for example, cannot possibly tear down the whole system. Malfunctioning device drivers can at most affect applications depending on them. Furthermore, they can simply be restarted. Additionally, due to their small size in the order of tens of kilobytes, microkernels can even be made formally verifiable to certain extent [21].

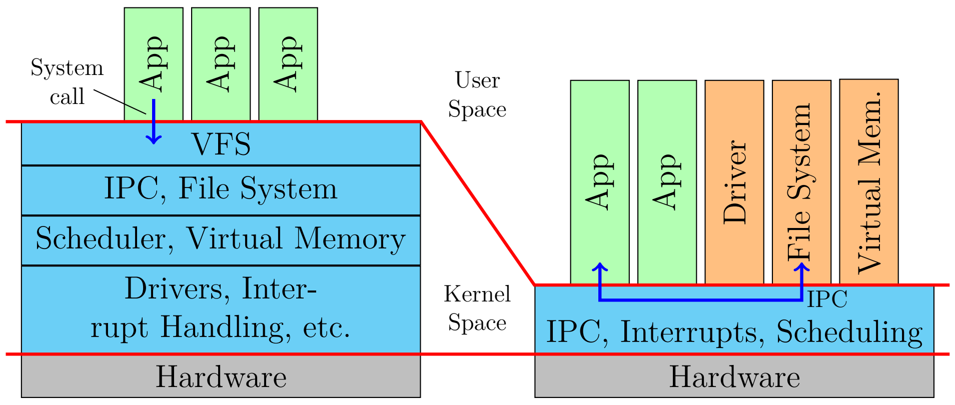

Figure 2.7.: Layer Diagrams of Monolithic Kernel Vs. Microkernel

On the other hand, microkernels still have not gained broad acceptance in the industry. An important cause for this is the belief that microkernels inherently introduce overhead into systems deploying this architecture. While monolithic kernels handle system calls in a single context, servicing an application in microkernel operating systems can cause multiple IPC jumps between server applications, introducing the feared overhead. Although microkernel researchers have shown that IPC overhead can be reduced to an absolutely acceptable minimum, acceptance is still only increasing slowly. [25, 24]

2.5.1. The NOVA Microhypervisor

The underlying microkernel used for this thesis project is the *NOVA* microkernel. NOVA is a recursive abbreviation for *NOVA OS Virtualization Architecture*. It was initially developed at TU Dresden and is now maintained at Intel Labs.

Motivation to develop NOVA arised from the observation that, just like monolithic operating systems in general, type 1 hypervisors have an ever growing TCB. The novel approach of NOVA is to merge microkernel design principles with hardware virtualization capabilities of modern processors. This way virtualization goals like the performance from existing virtualization solutions and the stability and security of microkernels could be combined. Because of this, NOVA is called a *microhypervisor*.

Mixing hardware virtualization features with microkernel design principles introduces a certain divergence from other virtualization solutions in terms of notation. Usually the terms *hypervisor* and *Virtual Machine Monitor* can be used interchangeably. In this case the microkernel is the hypervisor, hence the new term microhypervisor. But then NOVA only provides interfaces for use of the hardware virtualization features Intel VT-x and AMD-V, without actually virtualizing a whole computer architecture. The virtual architecture needs to be organized and controlled by a VMM which manages interaction between VMs and the physical resources of the host. Therefore, VMMs are their own user space application instances in isolated protection domains, each hosting a VM.

A NOVA system running VMs in different protection domains looks like a typical type 1 hypervisor, comparable to Xen. In contrast to existing solutions, the microhypervisor consists of about 9000 LOC[3] which are executed in kernel space. This is in orders of magnitude less privileged and trusted code. To stay at such a minimum of code size, NOVA provides only IPC mechanisms, resource delegation, interrupt control, and exception handling. The set of hardware being driven by the microhypervisor is minimized to interrupt controllers, the MMU and IOMMU.

An innovative key feature of NOVA is its *capability* based interface for providing applications access to resources. A capability stands for some kind of kernel object and is represented by a *capability selector*. Such a capability selector is an integral number, similar to UNIX file descriptors. Every protection domain within the system has its own capability space, which in turn means that no global symbols can exist to reference kernel objects. The capability paradigm enables fine-grained access control in accordance with the wish of designing a system with the principle of least privilege among all components [23].

NOVA implements only five different types of kernel objects:

Protection Domain PDs provide spatial isolation and act as resource containers consisting of memory space, I/O space and capability space.

[3]LOC: Lines Of Code

Execution Context An EC can represent a virtual CPU or an activity similar to a thread.

Scheduling Context SCs couple a time quantum with a priority. An EC has to be equipped with an SC to actually obtain CPU time for execution, i.e. only the combination of an EC with an SC can be compared to the classic concept of a thread.

Portal Portals are dedicated entry points into PDs for IPC or exception handling. Other PDs can be granted access to these.

Semaphore Semaphores provide a synchronization primitive between different ECs which can potentially execute on different processors. They are often used to signal the occurrence of HW interrupts or timer events to user applications.

Scheduling is implemented in form of a preemptive priority-aware round-robin scheduler. An execution context calling a portal can either be paused to not lose CPU time during portal handling or it can donate a part of it to the portal handling EC. This has in consequence that server applications could be implemented without pairing any of their ECs with SCs. In this case, the service would completely run with CPU time donated by portal callers. [34, 33]

2.5.2. The NOVA UserLand (NUL)

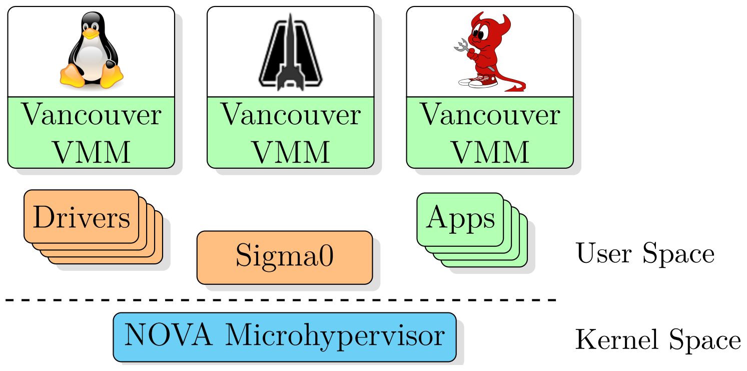

Figure 2.8.: System Structure of a Virtualization Platform Using NOVA/NUL

With a kernel whose size is minimized to extreme extent, the major part of the operating system runs in user space. Apart from the microhypervisor, the virtualization layer is decomposed into a root partition manager, device drivers, general services, and multiple VMM instances. NOVA was published together with the *NOVA UserLand* (NUL), which implements these parts.

The first application to be started after boot which NUL provides is the *root partition manager* called *Sigma0*[4]. Initially, it claims the whole memory and I/O

[4]Microkernel literature usually refers to σ_0 as the root partition manager process

resource area. In the following it creates protection domains etc. for other processes and *grant* or *delegate* them access to kernel objects like memory or I/O resources.

Figure 2.8 presents the composition of a booted NUL system. On top of NOVA, the Sigma0 process manages the access to system resources by the other processes. Running processes are applications, drivers and VMMs.

The Vancouver Userlevel-VMM

The VMM runs on top of the microhypervisor and as a user space application, it has its own address space. It supports the execution of unmodified operating systems in a VM, all implemented with about 20K LOC. The VMM is the handler for all VM exit events and contains one handler portal for every possible event type. As guests are usually preempted by VM exits when they try to execute privilege-sensitive operations, the VMM implements an instruction emulator to emulate the effect of those.

Guest-physical memory is managed by mapping a subset of the memory of the VMM into the guest address space of the VM. Memory mapped as well as port programmed I/O is either intercepted or directly mapped to real hardware. This way devices can either be emulated or real devices are passed through to the VM.

The software architecture of the VMM from a programmer's point of view tries to model a real computer system. A main class exists representing a *motherboard* containing a number of virtual *busses* via which system components can be connected. VM configurations are assembled from command line arguments. The respective command line argument handler dispatches the right object creation procedure for every argument item. Upon device model creation, the model object is connected to all relevant busses. Information like discovery messages, interrupt requests, network packets, etc. can then be sent over the according bus structure without knowing the recipients.

3. Design

This chapter describes the problem of live migration between heterogeneous hosts in detail and outlines the set of possible solutions. After discussing the advantages and disadvantages of each, the solution of choice will be backed with arguments and planned for implementation.

3.1. VM Migration between Heterogeneous Hosts

Although live migration of virtual machines is very useful and already supported by many industrial hypervisor solutions, it narrows down the variety of possible VM configurations. In scenarios where VMMs pass through real hardware to their VMs for performance reasons, live migration becomes very challenging. Since the VMM has no control over pass-through devices, it also has no knowledge about the state the device operates in. This complicates obtaining the device state to restore it on the target host. Another problem arises if the target host does not provide the same type of hardware device the VM had access to on the source host. In other cases the CPU model might slightly differ. The CPU of the target host might for example not provide support for SIMD instructions which were in active use by the VM on the source host.

Host-heterogenity therefore means for this project that both hosts which participate in a live migration have the same architecture in general, but they differ in important other parts. This thesis assumes that these parts can be peripheral PCI devices which are different models or do not exist in the other host at all.

A live migration solution being able to move VMs employing pass-through hardware devices obviously must be able to deal with this. To limit the complexity of the project, only network interface cards as differing pass-through peripheral components are regarded. These are involved in the most typical use case for pass-through devices in virtualization: Virtual machines with need for very high network throughput in data centers. As the platform in use implements faithful virtualization, guest code van be left untouched.

3.2. Possible Solutions

Researchers have proposed different solutions for the underlying problem. Pan et al. demonstrated how to migrate VMs using a real NIC by restoring its state. They worked around the problem of migration-unfriendly hardware by identifying typical

I/O operation sets which can be tracked on the source machine and then replayed on the destination machine. Access to these needs to be trapped by the hypervisor or may be logged by the guest driver itself. Consequently, the live migration host switch phase must not occur during such an operation set, rendering it a critical section. This constraint brings up the need for an additional synchronization mechanism between guest and hypervisor. Another consequence is that guest code, at least driver modules, has to be changed, which breaks with faithful virtualization. *Read-only* registers like statistic registers are plainly virtualized in their approach. Their *CompSC* framework providing all needed mechanisms for this general approach, needs to be ported onto any new kind of hardware device model to be used with it. Specific network interfaces appear to be rather simple to migrate due to a small set of operation sets, but other hardware like e. g. graphic acceleration cards proved to be very complex in this regard. However, one big advantage of this approach is that the VM has nearly no NIC throughput degradation during migration. Migrating a virtual machine to a host which can provide the guest with a pass-through NIC, but only a different model, is still not possible. [29]

Two different solutions assuming migration-unfriendly hardware, use proxy drivers in the guest operating system. Kadav et al. presented a *shadow driver* approach in which they implemented a logical network device driver between the real NIC driver and the networking stack of the guest operating system. This shadow driver records all I/O needed to direct the NIC on the destination host into the state its pendant at the source host was operating in. Triggered by an external migration event, the shadow driver can react to a host change and replay the device state [19]. Although this solution does again change the guest operating system source code, it does not touch the code of the device driver modules. Actually it is relatively device agnostic. If the recorded I/O was translatable to a semantic level, it might be possible to drive any NIC the target host provides. This idea might also be applicable to the CompSC framework.

A similar solution which does not depend on any guest code change was presented by Zhai et al. They also demonstrated the use of a proxy driver. Rather than implementing a new shadow driver, they made use of the existing *bonding* driver within the Linux kernel. The bonding driver is able to act as a logical NIC and to switch between different real NIC drivers in the background. Equipped with a real and a virtual NIC, a VM can be kept online after hot unplugging the real NIC at the source host, bringing the VM into an easily migratable state [36]. An interesting point about this solution is that it still works even if the target host may provide a different NIC model or does not have any pass-through NIC available at all. Furthermore, no knowledge about the individual NIC is required. The guest kernel has to provide drivers for all possible NIC models and needs to be configured for bonding device use. This work was committed to the Xen project. To actually use this feature, administrators have to write their own pre- and postmigration scripts.

3.3. Design

From all possible solutions, the ACPI hot plugging variant presented by Zhai et al. [36] is chosen as the main inspiration for the underlying project design. The most important reason for this choice is the constraint of faithful virtualization. Hot plugging is the only way to live migrate pass-through devices without changing any guest code. The bonding driver does already exist in the Linux kernel since version 2.0, which has been released more than a decade ago. Furthermore, this solution is absolutely NIC-model agnostic. It will be possible to unplug a device from vendor *A* and replug a device from vendor *B*. The bonding driver can easily handle such a series of events.

Being hidden behind standardized hot plug events, both the virtualization and live migration architecture do not have to be minded by the guest operating system developers/administrators. In general, guest operating systems solely need to support ACPI hot plugging as well as providing a driver similar to the bonding driver from the Linux kernel. Administrators can then just *configure* the system to be able to make best use of a fast, sometimes *disappearing*, pass-through NIC and a slower, but always available, virtualized NIC.

A downside of this approach is that the VM which shall be migrated will suffer of network throughput degradation during migration, which might be a problem in special use cases. On the contrary, it might be advantageous to throttle guest networking anyway, since very high guest network traffic will inevitably increase the writable working set (WWS). Other live migration solutions intentionally throttle the network throughput of the guest to both reduce the WWS and to increase the throughput for host packets [6].

The VMM application running in the NUL user space environment on top of the NOVA microhypervisor is able to live migrate VMs which are not configured for network use. For this reason, the virtual NIC model needs to be extended with live migration support at first. This extends the area of deployment of the live migration feature to VMs providing services over network. To be able to initiate and control hot plugging features like real server hardware does, an ACPI controller model is required. As the live migration procedure shall automatically deal with pass-through devices, it needs to be extended to actually use these new features of the VMM.

The next subsection describes the adapted live migration strategy. Thereafter, the necessary changes to make the NIC model migratable are outlined. All needed interfaces and the range of responsibilities of the ACPI controller model are determined at last.

3.3.1. The Adapted Live Migration Process

The existing live migration algorithm of course needs to be extended to support the new migration strategy enabling it to handle pass-through devices. Figure 3.1 gives a high level overview on the extensions to the migration procedure:

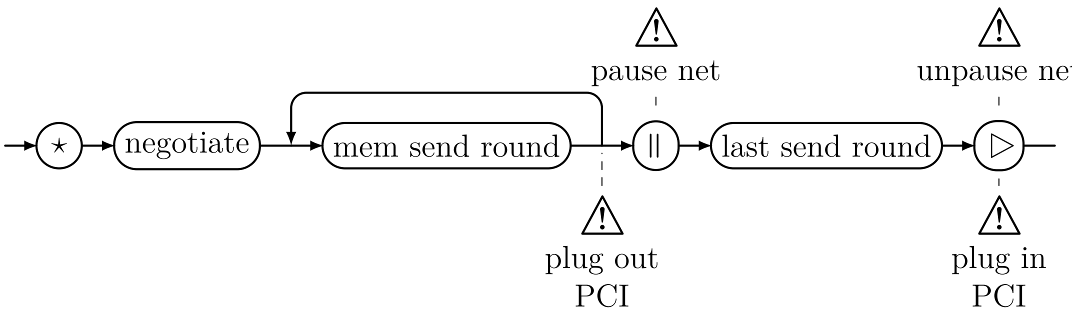

Figure 3.1.: Overview of Live Migration Process Phases and Needed Extensions

1. At first, the live migration procedure is *initiated* by the user or an algorithm.
2. The following *negotiation* stage allows to check if the destination host supports pass-through migration by giving VMs access to such devices. The user or an algorithm might accept the migration of specific VMs only if the other side supports pass-through devices. The migration might then otherwise be aborted to look for another destination host, for example. Decisions of this kind should happen at this stage. However, if the migration was accepted after negotiation by both sides, resources are claimed at this stage.
3. After negotiation, the whole guest memory range is synchronized between source and target, which is done during multiple *send rounds*, depending on the workload of the VM.
4. As the last stage on the source host, the VM is frozen to send remaining memory differences as well as the states of all devices and the VCPU to the destination host. PCI devices need to be unplugged at some point in time *before VM* freeze. This is very important, because the guest OS has to participate in the unplugging procedure. Choosing the right moment for unplugging is not trivial, as will be explained later.
5. As the virtual network interface model that is still running continuously dirties guest memory when receiving packets, it needs to be paused as well.
6. After transfering the remaining state data, the VM execution is resumed on the destination host. The virtual network device can be unpaused at the same time. Replugging an available passthrough-device can also happen immediately.

While the right moment of virtual NIC model unblocking and replugging of PCI-devices is clear, especially the right moment for PCI *un*plugging is not. Since the reaction time of the guest OS to the unplug event issued by the VMM has to be respected, this cannot just be done synchronously before guest freeze. Therefore, PCI unplugging has to take place during the second last memory resend round at the latest. The challenge occuring here is that it is hardly possible to determine which memory resend round this might be. As the guest workload may change during migration, it is also not possible to tell how many resend rounds will be needed. The decision about when to freeze the VM to stop the round based resend strategy is usually decided considering network throughput-rate and dirtying rate, which can be calculated *after* any resend round.

However, a pass-through device used by the VM, driven to provide networking with maximum throughput, might boldly increase the WWS. Early unplugging of pass-through devices would force the VM to switch over to its leftover virtual NIC. As soon as the guest uses its virtual NIC, the VMM gains control over the network throughput-rate of the guest machine. Being generally slower, forced use of the virtual NIC effectively lowers the size of the WWS. Furthermore, the VMM could additionally *throttle* the virtual guest NIC to increase this effect. Another important detail is the missing support for tracking page dirtying via DMA initiated by devices. Unmapping all pass-through devices at the beginning of the migration process will work around the problem of possibly undetected dirty memory regions which were manipulated by DMA. Therefore, the beginning of the live migration procedure is chosen as the moment in which pass-through devices shall be unplugged.

3.3.2. Virtual NIC Migration

As pass-through devices are considered unmigratable in this project, the guest has to use a virtual NIC during the process of live migration. The virtual NIC model provided by the underlying VMM represents an Intel 82576 VF network controller in form of a PCI card. Live migration has been developed much later than the device model, hence it did not support this feature, yet.

To support live migration, the state of the frozen device model has to be read out and subsequently written back to its counterpart on the target machine. This is usually trivial, because the state of a frozen model does not change and consists of virtual registers which are all represented by easily readable memory fields in the memory space of the VMM. In the underlying VMM application, the device model is only accessed by a worker thread handling incoming network packets as well as the VM accessing registers via I/O read and write operations. Blocking both the VM and the network worker thread therefore leaves the device model in a frozen state. VM blocking was already implemented. Network worker thread blocking just involves a suiting kind of lock/semaphore, since I/O worker threads usually loop over obtaining and processing portions of work. A model state which was read out of a device model and then restored to another device model on a different host can seamlessly continue its service after being released from freeze.

The special case of migrating a network device introduces another problem which does not occur with non-network devices. While using the freshly reactivated virtual NIC model for sending packets would not introduce any problems, receiving them would: Packets would simply not be forwarded to the destination host on which the device now operates. They are still routed to the old host which processed them before the migration took place. After migration, all network hosts and network switches in the network need to be informed about the new route packets have to be directed over. Advertising a new MAC address which an IPv4 address shall be bound to is done with *Address Resolution Protocol* (ARP) packets. For IPv6 networks, a similar mechanism called *Neighbor Discovery Protocol* (NDP) exists. This thesis assumes IPv4 networks. ARP in combination with DHCP is generally used for IP address negotiation for new hosts entering an IP network. To update the according ARP table entry of all other hosts and switches, a *gratuitous ARP* packet has to be sent. An important detail is that the MAC address does not change, although gratuitous ARP is designed for this case. The packet the VMM has to send contains information (guest IP and guest MAC address) the other hosts already know about. Actually no ARP table entry will be changed by receiving hosts. However, network switches will notice that this packet arrived via a different port than the port they would expect a packet from, sent by this host. In consequence, all receiving switches will update their internal routing tables. From this moment on, packets are correctly routed to the migrated VM. A situation in which this mechanism does not work are e. g. networks where administrators block such packets or configure switches to ignore them. The motivation behind this is security, since gratuitous ARP can also be used by third parties to reroute traffic in order to be able to monitor and/or even manipulate it.

In this situation the VMM is sending ARP packets with both IP and MAC address of the guest. This implies that the VMM has to obtain this information from within the guest. An obvious way is inspecting packets the guest has sent out to the network prior to its migration. As the guest is using a virtual NIC in the last migration stage, the VMM has control over all guest packets and can easily extract all needed information from them.

3.3.3. ACPI Controller Model

The VMM has to command the guest operating system to unplug pass-through devices before migration, and to plug them in again after migration. Because of the host-guest interaction this sounds very similar to paravirtualization. But in fact, it is still faithful virtualization, because this particular interface was designed without virtualization in mind. In order to be able to express these commands, the VMM has to implement the hardware side of the corresponding ACPI interface. The ACPI subsystem communicates with the operating system via tables located in system memory and via interrupts combined with reads from and writes to I/O registers. By referencing each other, the tables represent a tree structure. This tree contains both static and dynamic parts. The static parts describe characteristics of the system to the guest OS. To be able to drive more complex subsystems via standardized interfaces, ACPI provides fields filled with bytecode within the dynamic parts of the tables. Mainboard manufacturers use this to define a tree with multiple scopes of system properties and device descriptions which can even contain program routines the OS can interpret and execute.

ACPI features in the underlying VMM have only been implemented to an extent where a minimal set of ACPI tables provides the guest operating system with information about the PCI configuration space, local APICs, and the power management timer. To support ACPI hot plugging, general ACPI support has to be provided to make Linux activate its ACPI mode at boot. Necessary for this to happen is the existence of specific *power management* and *general purpose event* bit field register sets. These fields can be configured to reside in programmed I/O or memory mapped I/O space using the respective entries in the static ACPI tables. Furthermore, it is necessary to provide a table entry informing about the interrupt number *System Control Interrupts* (SCI) are assigned to, when issued by the ACPI controller.

As soon as the guest Linux kernel activated its ACPI mode, it can communicate with the ACPI controller. Therefore this controller has to be hooked into the existing I/O bus to intercept reads and writes which are directed to ACPI related registers. To support hot plugging, the controller needs to manage special internal state register variables for this purpose. As the guest operating system has no *a priori* knowledge about hot plugging related registers, it needs to be educated about them. This is achieved by byte-encoded structure and procedure descriptions in the dynamic part of the ACPI system description.

In the end, the ACPI controller will just trigger general purpose events without any knowledge about their semantics. The bytecode in the dynamic system description tables gives specific general purpose event bits the semantic meaning of hot plugging, etc. Other virtualization solutions like Xen and Qemu are able to generate proper bytecode prior to booting a new VM. The tables are then configured individually to allow hot plugging for the number of currently configured virtual PCI ports. To not exceed the complexity of this prototype, the controller will be able to deal with the hot plugging of only a fixed number of PCI slots.

The controller will then have to provide an interface for being triggered by the migration code to initiate hot plug events. This interface has to simplify the mapping of a PCI device, regardless of being a virtual one or a pass-through one, to the respective event.

Furthermore, the final ACPI controller model will be a device model with register states. Consequently, it needs to be embedded into the migration algorithm as both part of it and part of the set of migratable device models.

This chapter illustrated possible solutions for the thesis problem and outlined how ACPI hot plugging can be used to make VMs utilizing pass-through devices migratable in a very versatile way. Necessary changes to the existing NIC device model and the need for a new ACPI controller model were identified and their extent specified. The next chapter shows how the design was implemented into the existing VMM code base.

4. Implementation of Pass-Through-Device Migration

After discussing possible design ideas in the last chapter, the implementation of the design decisions made will now be described. At first the relevant parts of the existing live migration architecture are illustrated to enhance the understanding of how they were extended to support the new feature. Then the new subsystems are explained one by one. These are the extensions to the virtual NIC model to make it migratable, the ACPI controller model and all adaptions which had to be done on the VMM. In the end explaining the Linux bonding driver module and how guest systems can be configured to make use of the new VMM-feature, the perspective of the guest is covered.

4.1. The Existing Live Migration Infrastructure

The underlying live migration feature is the product of an internship which was finished just prior to the thesis project. Its most important parts are its *memory synchronization mechanism* as well as the *restore bus* it mainly uses to cover device model state transfer. This section will explain these in detail after describing the general migration process.

Figure 4.1 gives a high level overview on the different phases of the live migration algorithm as it existed prior to beginning the thesis project. The live migration procedure is encapsulated in its own class, which is allocated in memory as soon as the user commands the VMM to migrate its guest system to a given destination host. Subsequently, source and destination host negotiate if the VMM configuration in use can be started on the destination host. If starting an empty VMM with this particular configuration succeeds, the destination host tells the source VMM on which TCP port it is listening. Directly after connecting to the new listening VMM, the source VMM initially transfers the whole guest memory state. Due to changes of the memory state caused by the guest, the memory synchronization subsystem tracks all dirty memory pages. After the first transfer, the VMM resends all dirty pages. This is done roundwise with the aim to incrementally narrow down the list of dirty memory pages towards the end. As soon as the rate with which the guest dirties its memory rises above the network throughput-rate, the dirty memory list size converged to its practical minimum. This is called the *Writable Working Set* (WWS). At this stage, the guest is frozen to transfer the rest of dirty memory pages together with all device model states. Guest execution is then continued on the

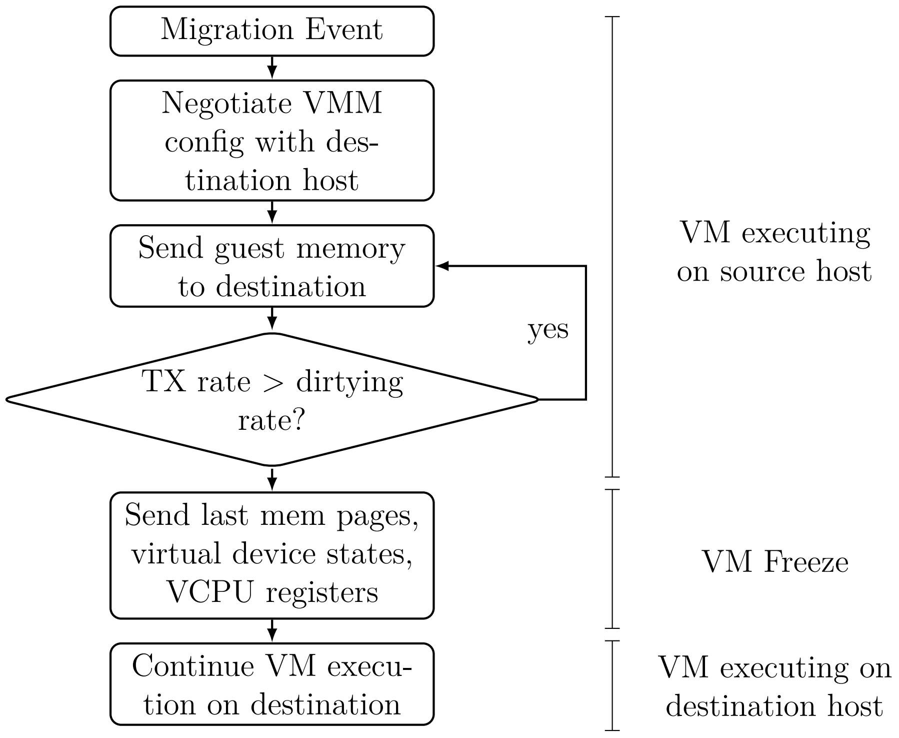
Migration Event
Negotiate VMM config with destination host
Send guest memory to destination
TX rate > dirtying rate?
yes
Send last mem pages, virtual device states, VCPU registers
Continue VM execution on destination
VM executing on source host
VM Freeze
VM executing on destination host

Figure 4.1.: Workflow of the Migration Algorithm

destination host. This memory synchronization strategy is also referred to as the *pre-copy* strategy [6].

4.1.1. Memory Synchronization

Synchronizing the guest memory state between source VMM and destination VMM while the guest is running, is the most important part of the live migration feature. In order to know which parts of guest memory have been changed since some specific point in time, the VMM has to be able to record write attempts of the guest system to its memory. Recording write attempts is done by exploiting the *extended paging* mechanism which is provided by the hardware.

By removing the *write* access right from a guest memory mapping in the extended page table, it is mapped read-only within the guest. While this is transparent to the guest operating system, it works like a trap when the guest tries to write to an address in this mapping. In reaction to the then occuring VM exit, the NOVA kernel will call the *EPT Violation* portal the VMM provides. This portal is comparable to a usual page fault handler and is supposed to return the corresponding host-virtual address for this guest-physical mapping. This mapping answer is encoded as a *Capability Range Descriptor* (CRD) which contains information about the page quantity and offset as well as access right bits. In NUL, the VMM generally maps the guest memory range twice. The first mapping is located somewhere accessible for the VMM, while the second mapping, which is also accessible by the guest, is located at the lowest addresses.

Combined with the system calls `nova_lookup` and `nova_revoke`, this mechanism can already be used to completely track the memory writing behavior of the guest system. In the VMM, all OS-specific code is kept within the main VMM class source file, which reduces the necessary effort for porting it to other operating systems. Therefore, objects within the project which need to use system-specific functions, have to call the main class which implements them in an OS-specific way. This has to be extended by a function returning guest-write-enabled page ranges from the guest memory space.

```
Crd next_dirty_region()
{
  static unsigned long pageptr = 0;
  unsigned long oldptr = pageptr;

  // The number of pages the guest has
  const unsigned physpages = _physsize >> 12;

  /* If this was called for the first time,
   * the page tracking mechanism is now activated.
   * The consequence of this is that all new mappings
   * will be done in 4K page size granularity. */
  _track_page_usage = true;

  Crd reg = nova_lookup(Crd(pageptr, 0, DESC_MEM_ALL));

  while (!(reg.attr() & DESC_RIGHT_W)) {
    /* Fast-forward the pointer to the next
     * RW-mapped page. */
    pageptr = (pageptr + 1) % physpages;
    if (pageptr == oldptr)
      /* We traversed the whole guest memory
       * space once and did not find anything. */
      return Crd(0);
    reg = nova_lookup(Crd(pageptr, 0, DESC_MEM_ALL));
  }

  /* reg now describes a region which is guest-writable.
   * This means that the guest wrote to it before and it
   * is considered "dirty". Make it read-only, so it is
   * considered "clean" again and return this range. */
  Crd host_map((reg.base() + _physmem) >> 12, reg.order(),
    DESC_RIGHT_W | DESC_TYPE_MEM);
  nova_revoke(host_map, false);

  pageptr += 1 << reg.order();
  if (pageptr >= physpages) pageptr = 0;

  // The returned CRD contains size and offset of this mapping
  return reg;
}
```

Listing 4.1: The Function Returning Dirty Guest Memory Ranges

Code Listing 4.1 shows the implemented code. The `next_dirty_region` function is a state machine which takes no input variables. It maintains its own static memory pointer in form of `pageptr`, which holds a page number. This page pointer is set to the next RW-mapped page range offset on every call. The `nova_lookup` system call returns a CRD describing the range which is mapped to this page offset. If no mapping has been applied, the CRD is empty. Each call returns another RW page range and remaps it read-only, so it will immediately trap the guest on its next write attempt. Remapping is done by calling `nova_revoke` on a memory range with an input CRD describing the range with all access bits to be removed. As soon as the running page pointer exceeds the total range of guest page frames, it is wrapped over and starts counting from address 0 again. If no page was written since the last call, the function will recognize that it moved the page pointer once around the range and returns an empty CRD. The `_track_page_usage` variable is set to tell the EPT violation portal handler which maps memory for the guest, that it shall do mappings in page size granularity.

Memory can also be dirtied by DMA. Monitoring memory accesses of this kind is currently not implemented. This is not critical at this stage, since the only devices which would use DMA are pass-through devices. These are unplugged during migration before sending guest memory over network.

4.1.2. The Restore Bus

The `DBus` class[1] plays a very central role in the software architecture of the underlying VMM. An instance of `DBus` represents a communication bus and keeps a list of `Device` class instance pointers. Any object having access to such a `DBus` instance can send *messages* over the bus to reach all connected objects. A central instance of the `Motherboard` class maintains several `DBus` objects. Any object having access to this `Motherboard` class instance can reach semantically bundled lists of devices.

On startup of the VMM, handlers for every type of command line parameter are called. Device parameter handlers instantiate the respective device class and provide it with a reference to the motherboard object. Equipped with this, devices can connect themselves to the bus structures they make use of. This kind of architecture tries to model the interconnect of hardware components like in physically existing computers.

Device class derivatives have to additionally derive from the class `StaticReceiver` if they need to be reachable via busses. `StaticReceiver` provides a type-agnostic interface for being called from the bus user. The method to be called via the bus structure which needs to be implemented is illustrated by an example in Code Listing 4.2.

[1]This has nothing to do with the *Desktop-Bus* from the freedesktop.org project.

```
bool  receive(MessageType &msg) {
  if (msg.type != MessageType::MY_TYPE) return false;

  /* process msg */

  return success;
}
```

Listing 4.2: An Example Bus Receiver Method

With the migration feature, a new bus was introduced. The *restore bus* in combination with messages of type `MessageRestore` is used for any migration-related communication. Restore messages can be used to *obtain* or *restore* the state of a device. They contain type fields which can be used to identify the message recipient. Another field within this message type tells if the receiving device shall copy its state into it or overwrite its state from it. Keeping device-dependent state serializing/deserializing code within each migratable object makes the rest of the migration code more generic and device-agnostic.

Apart from serializing/deserializing device states, the restore bus can be used for other purposes. One example is the VGA device model, which can also receive restore messages at the beginning of a live migration at the receiver side to switch to the correct video mode. This way the screen would not display garbage because the frame buffer is already restored, while the VGA device model is still waiting for its new state. The restore bus has proved to be a handy interface for generic pre- and post-migration code execution.

Listing 4.3 shows a minimal migratable device model example which can be added to the system at VMM launch. Its class inherits from the class `StaticReceiver` to be able to receive messages in general. The parameter handler code macro in Line 67 which is called during VMM initialization if the `devicemodel` command line parameter was used, hooks it into the restore bus.

```
 1 #include "nul/motherboard.h"
 2 
 3 class DeviceModel : public StaticReceiver<DeviceModel>
 4 {
 5   private:
 6   mword _register_a;
 7   mword _register_b;
 8   /* ... */
 9 
10   bool _processed; // Restore state
11 
12   public:
13   /* ... */
14 
15   bool receive(MessageRestore &msg)
16   {
17     /* How many bytes does the serialized device
18      * state consume? */
```

```
    const mword bytes =
       reinterpret_cast<mword>(&_processed)
      -reinterpret_cast<mword>(&_register_a);

    /* The restart message is sent as a broadcast
     * before reading/restoring the devices and
     * collects the combined size of all serialized devices
     * in its msg.bytes field. */
    if (msg.devtype == MessageRestore::RESTORE_RESTART) {
      _processed = false;
      msg.bytes += bytes + sizeof(msg);
      return false;
    }

    /* Skip if this message is for another device type or
     * this device was serialized already. */
    if (msg.devtype != MessageRestore::RESTORE_DEVICETYPE ||
        _processed)
      return false;

    if (msg.write) { /* serialize and save */
      msg.bytes = bytes;
      /* The message sender guarantees that msg.space,
       * a byte field, is large enough. */
      memcpy(msg.space,
             reinterpret_cast<void*>(&_register_a),
             bytes);
    }
    else { /* deserialize and restore */
      memcpy(reinterpret_cast<void*>(&_register_a),
             msg.space, bytes);
    }

    /* The device is now processed and will ignore further
     * messages with exception of the restart message. */
    _processed = true;

    return true;
  }

  DeviceModel(Motherboard &mb)
  : BiosCommon(mb),
  _register_a(0), _register_b(0), _processed(false)
  { }
};

PARAM_HANDLER(devicemodel,
  "devicemodel - command line parameter to add a new devicemodel")
{
  DeviceModel * dev = new DeviceModel(mb);
  /* ...*/
  mb.bus_restore.add(dev,
    DeviceModel::receive_static<MessageRestore>);
}
```

Listing 4.3: Minimal Migratable Example Device Model

4.2. Network Interface Migration

The existing virtual Intel 82576 VF network controller was not within the set of migratable devices, yet. This controller provides *Single Root I/O Virtualization* (SR-IOV), which means that the device is capable of multiplexing itself. Thus, the device model imitates only a functional part of the device. Such a functional part, i.e. a *virtual function device*, provides 2 receive and 2 send queues as well as a set of memory mapped registers [16, 22]. In the model of this device function, these are represented by plain memory ranges in the address space of the VMM and can easily be read out and overwritten via the restore bus like any other device model.

Packet receiving is handled by a dedicated thread within the VMM. This networking thread blocks on a semaphore which is counted up whenever a network packet arrives. Raw network packets are processed in the NIC model with the scheduling context of the network thread. This means that the device state as well as guest memory may be changed from within the VMM even when the VM itself is frozen. As guest memory and device states must remain unchanged during the freeze period, the receiving of packets for the VM must be stopped. This is achieved with a new type of `MessageHostOp` for the host operation bus. The host operation bus exists to encapsulate any type of operating system specific code into the main class. For pausing guest networking, a boolean variable called `_network_paused` was added to the main class. The host operation of type `MessageHostOp::OP_PAUSE_GUEST_NETWORK` simply toggles this bit. Whenever it is set to *true*, packets which are received for the guest, are dropped. Dropping is not critical for the majority of use cases, since higher networking layers like TCP will simply resend them.

After restore at the destination host, the NIC model can immediately be fed with received packets again. For the freshly resumed VM it is fully functioning and can send packets from the guest into the network like on the source host before. The last problem to solve is that the other network participants as well as the network switches have no knowledge of the migration of the guest. All network switches continue routing packets to the source host. The VM will not receive any packets at this time. After the first connection times out, hosts will begin to send ARP discovery packets, which will solve the problem. Waiting until the first timeout occurs would disrupt all network services on this VM. This condition is not acceptable, because the host would then appear to be down although it is actually running and able to continue servicing requests over network.

4.2.1. Keeping TCP Connections Alive with Gratuitous ARP

To accelerate the rediscovery of the moved VM in the network, the VMM can propagate the new position of the VM to all network participants. Since this VM is faithfully virtualized, it cannot do this itself due to the lack of knowledge about its own virtualization, and hence also its physical movement. Just after the device restore and still before continuing VM execution, the VMM sends a so called *gratuitous ARP* packet. Of course, gratuitous ARP does only work if the VM was moved within the same broadcast domain. In IPv6 networks the *Network Discovery Protocol* (NDP) would be used instead of ARP.

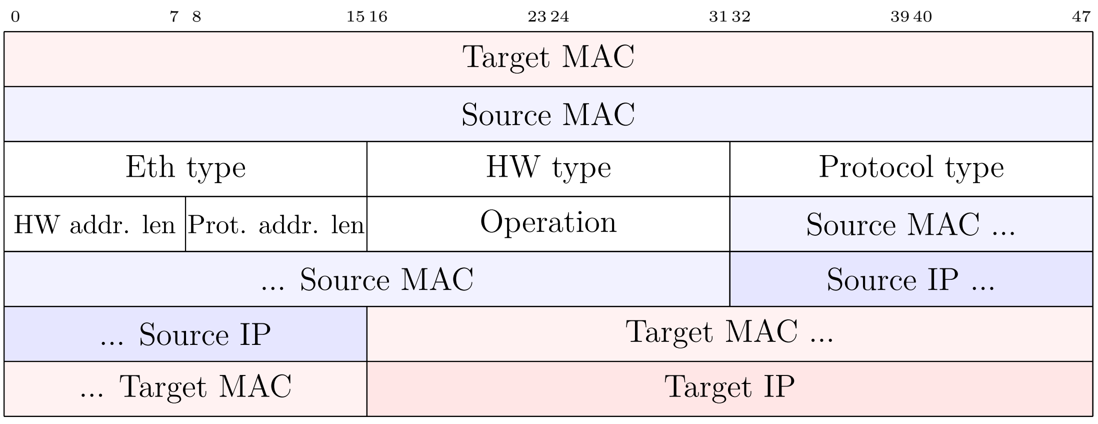

Figure 4.2.: Diagram of a Raw ARP Packet

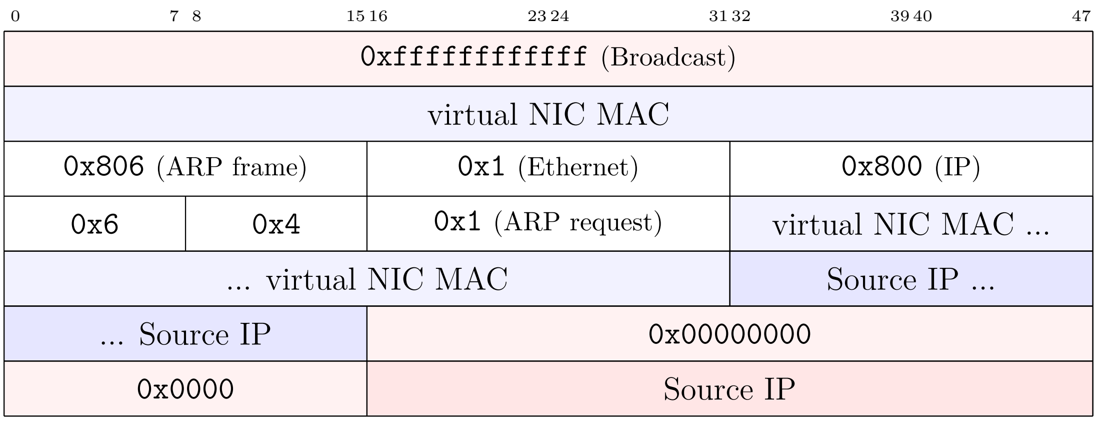

Figure 4.3.: Gratuitous ARP Packet Filled with Values

Figure 4.2 illustrates the content of an ARP packet and Figure 4.3 shows it filled with the values it is actually sent out with. An ARP packet qualifies as *gratuitous* if the following conditions apply:

- The Ethernet target field is the broadcast MAC address.
- The operation field is `0x1`, which stands for *ARP request*.
- The ARP target field is zero.
- Both source and target IP address fields are the IP address of the VM.

A gratuitous ARP packet represents a request like "What is the right MAC address for this particular IP address?". When this ARP packet was sent by the destination VMM, the NIC model within the source VMM would be the only device to answer this ARP request. However, it is paused at this stage. No other host has this IP address and thus no answer is received. All hosts and switches update their ARP tables. From this moment packets are correctly routed to this destination again. TCP connections will immediately be back to service. Depending on how long the downtime of the VM lasted, this might stay unnoticed by users of network services.

To be able to send such a packet, the NIC model additionally needs to know the IP address of the guest. Furthermore, it needs the MAC address the guest uses, as this is not necessarily the MAC address the NIC model was assigned to. Since the VMM forwards all packets from the NIC model to the host NIC driver, it can inspect guest packets and extract both the IP and MAC addresses. Extracting these from packets requires that the guest was already using the network before its migration. If it has not sent any packets up to this moment, it will also not have any standing TCP connections which need to be preserved. The VMM does not need to send any gratuitous ARP packet in this case.

The ARP protocol does not use any authentication. This enables network participants to take over any IP address at any time, which is also known as *ARP spoofing*. Administrators who have the responsibility to keep computer networks secure, will often consider the use of this mechanism by third parties an *attack* and try to prevent this scenario. Common countermeasures are fixed ARP table settings on secured hosts, or network switches allowing MAC address changes only when a cable is also physically replugged, etc. Migrating the IP address will not work in such networks.

4.3. ACPI Controller Model and Hot Plugging

Just before the guest operating system can boot, the virtual BIOS of the VMM prepares and initializes the system. One part of this initialization code is writing ACPI tables into the guest memory. Since ACPI was only needed to that extent that the guest OS can inform itself about the PCI Express configuration space and LAPIC geometry, more tables had to be added first.

Figure 4.4 illustrates the organization of ACPI tables in a tree structure. The root of the tree is the *Root System Description Pointer* (RSDP). A booting guest OS usually scans for the magic string signature `"RSDP"` within the first KB of the

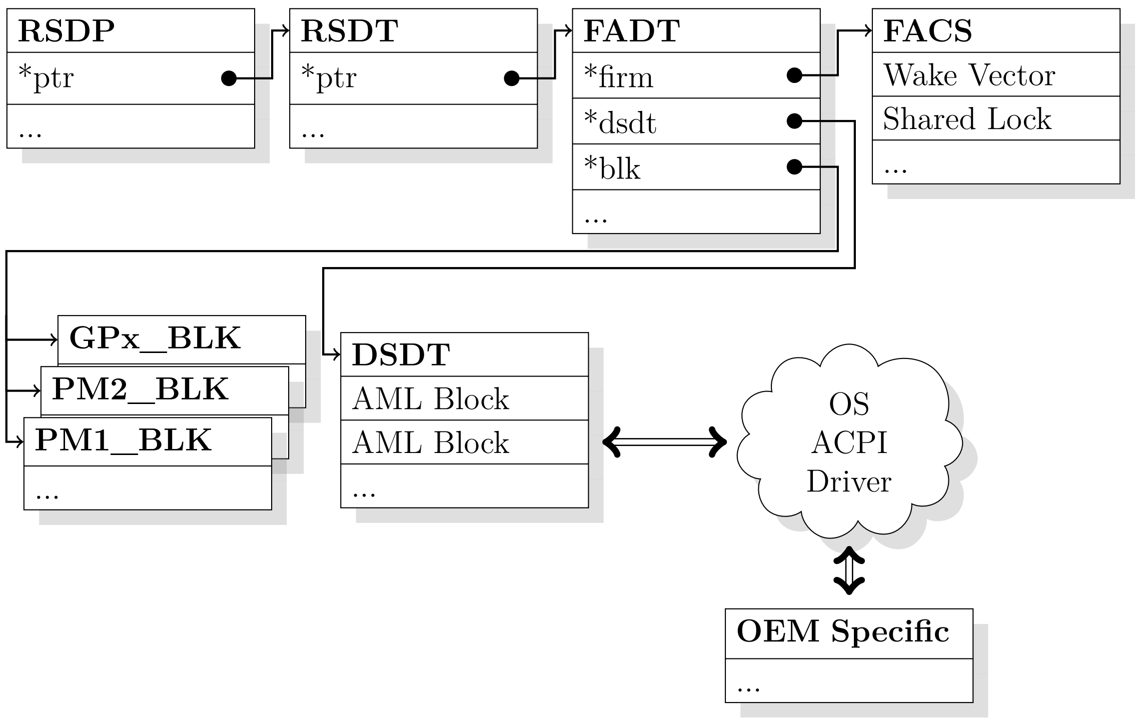

Figure 4.4.: ACPI Table Structure as Found in Memory at Boot

Extended BIOS Data Area (EBDA) and within the BIOS read-only memory space between the addresses `0xE0000` and `0xFFFFF`. The memory structure positioned just after this magic string contains a pointer to the *Root System Description Table* (RSDT), which in turn contains a pointer to the *Fixed ACPI Description Table*. The by then yet missing parts were the *Firmware ACPI Control Structure* table (FACS) and an entry within the FADT pointing to it. Also still missing were the *Power Management* (PM) and *General Purpose Event* (GPE) I/O register sets. Entries denoting address space, address, and size for the status, enable, and control registers of the so called *PM1a* and *PM1b* register blocks needed to be added to the FADT. The same applied to the status and enable registers of the *GPE0* and *GPE1* register blocks.

Adding new ACPI tables to the existing structures can be done via the *discovery bus*. This bus accepts messages of type `MessageDiscovery`, which transport a string identifier of the table to manipulate, a byte offset, a field length and the value to be written into this field. Any device model which needs to store information in memory before guest boot can be attached to the discovery bus. The virtual BIOS reset routine of the VMM sends an initial discovery message which triggers all connected device models to answer with messages to populate the memory with their table information. Thus, the discovery bus is the first bus the ACPI controller model needs to be connected to. Code Listing 4.4 shows the code with the necessary discovery message answers to bring the FADT into a state which is accepted by the Linux kernel. The helper method `discovery_write_dw` takes an identification string for the target table, a byte offset within it and a double word sized value and transforms it into a `MessageDiscovery` which is then sent over the bus. `"FACP"` is the identification string of the FADT.

The last three discovery write lines in the code listing have not been mentioned, yet. Newer systems leave these values in the FADT empty, which means that the hardware is always in ACPI mode rather than *System Management Mode* (SMM). If the `SMI_CMD` field is set to a port value and the `ACPI_ENABLE` as well as `ACPI_DISABLE` fields contain magic key-values, the guest OS will recognize this and try to switch the hardware from SMM to ACPI by writing the `ACPI_ENABLE` value to the `SMI_CMD` I/O port. The hardware has then to set the `SCI_EN` bit within the PM1a control register. This bit is the indicator for the guest OS that the transition from SMM to ACPI was successful. The ACPI controller model can then safely assume that the guest kernel activated its ACPI module.

```
bool  receive(MessageDiscovery &msg) {
  if (msg.type != MessageDiscovery::DISCOVERY) return false;

  /* The following FADT entries will tell the guest kernel
   * how to interact with the system when receiving
   * System Control Interrupts (SCI).
   * Only the GPE part is important for hot plugging, but
   * all the PM-stuff is mandatory for event management
   * to work.
   */
  discovery_write_dw("FACP", 56, PORT_PM1A_EVENT_BLK);
  discovery_write_dw("FACP", 60, PORT_PM1B_EVENT_BLK);
  discovery_write_dw("FACP", 64, PORT_PM1A_CONTROL);
  discovery_write_dw("FACP", 68, PORT_PM1B_CONTROL);
  discovery_write_dw("FACP", 88, PM1_EVT_LEN, 1);
  discovery_write_dw("FACP", 89, PM1_CNT_LEN, 1);

  discovery_write_dw("FACP", 80, PORT_GPE0_STATUS, 4); // GPE0_BLK
  discovery_write_dw("FACP", 84, PORT_GPE1_STATUS, 4); // GPE1_BLK

  discovery_write_dw("FACP", 92,  4, 1); // GPE0_BLK_LEN
  discovery_write_dw("FACP", 93,  4, 1); // GPE1_BLK_LEN
  discovery_write_dw("FACP", 94, 16, 1); // GPE1_BASE (offset)

  /* This is used at boot once. Linux will write
   * CMD_ACPI_ENABLE via system IO using port PORT_SMI_CMD
   * to tell the mainboard it wants to use ACPI.
   * If CMD_ACPI_ENABLE was defined as 0x00, the guest kernel
   * would think that ACPI was always on. Therefore, this is
   * optional and one could just erase the next three lines.
   */
  discovery_write_dw("FACP", 48, PORT_SMI_CMD);
  discovery_write_dw("FACP", 52, CMD_ACPI_ENABLE, 1);
  discovery_write_dw("FACP", 53, CMD_ACPI_DISABLE, 1);

  return true;
}
```

Listing 4.4: MessageDiscovery Receiver Method of the Implemented ACPI Controller

The PM1a and PM1b registers are not needed for hot plugging, hence they remain mainly unused in this case. Linux demands their existence before activating its ACPI module because of the `SCI_EN` bit within the PM1a register and the fact that SCI handling involves reading the PM and GPE registers to determine the event type which caused the interrupt.

To make I/O reads and writes from and to these registers within the ACPI controller model actually work, the controller class code needs to be connected to the virtual I/O bus of the VMM. I/O reads from the guest are processed by a `MessageIOIn` receive handler. All named registers and all registers to follow are read without any side effects. Thus, the I/O read handler just returns the register values, which are 32 bit unsigned integer variables.

```
bool  receive(MessageIOOut &msg) {
  switch (msg.port) {
  case PORT_SMI_CMD:
    /* During boot, the guest kernel checks PORT_SMI_CMD
     * in the ACPI FADT table. If SCI_EN is not set,
     * the system is in legacy mode. Hence, it sends the
     * CMD_ACPI_ENABLE cmd it got from the FADT again to
     * this port and then polls for SCI_EN until it is set.
     * ACPI is then officially active. */
    if (msg.value == CMD_ACPI_ENABLE) {
      _pm1a_control |= 1U; // Setting SCI_EN bit
    }
    else if (msg.value == CMD_ACPI_DISABLE) {
      _pm1a_control &= ~1U;
    }
    return true;

    /* Usual write-registers as well as
     * write-clear registers are handled here.
     * ...
     */
  }

  /* Deassert this IRQ if all enabled events were cleared
   * by the guest. This interrupt is thrown again otherwise. */
  if (!(_pm1a_status & _pm1a_enable) &&
      !(_pm1b_status & _pm1b_enable) &&
      !(_gpe0_sts    & _gpe0_en) &&
      !(_gpe1_sts    & _gpe1_en)) {
    MessageIrqLines msg(MessageIrq::DEASSERT_IRQ, 9);
    _mb.bus_irqlines.send(msg);
  }

  return false;
}
```

Listing 4.5: I/O Write Receive Method of the Implemented ACPI Controller

Code Listing 4.5 shows the implementation of the I/O write handler. Code handling read-write and write-clear registers is cut out for this listing. Lines 3 to 15 show the code which handles the transition from SMM to ACPI mode and back. The code lines from 26 to 32 are interesting in terms of interrupt delivery. Asserting an SCI interrupt to notify the guest OS about PM or GPE events is done by sending a `MessageIrq` over the IRQ bus of the VMM. This interrupt has to be deasserted explicitly after the guest has processed all events. Checking if the guest has done this, is achieved by logically *and*ing each status register with its companion enable register.

Being connected to the discovery and I/O in and out busses, the ACPI controller model is able to communicate with a Linux guest to make it setup its ACPI subsystem. This is the foundation for further development of hot plugging features.

4.3.1. Adding Hot Plugging

Although both the VMM and guest OS participate in hot plugging mechanisms, only VMM code had to be changed. These code changes can be split into two parts: The first part are extensions to the ACPI controller model in the VMM to enable it to play the host role of the hot plug workflow. More complex is the second part, which consists of an AML block added to another new ACPI table, the DSDT.

ACPI controller part

Beginning with the ACPI controller model part, an extension to trigger the initial hot plug event is needed. All hot plug related events, which are *unplug PCI card x* and *replug PCI card x*, are wired to the same GPE. Listing 4.6 shows the implemented method to trigger a GPE. Both GPE status registers share a bitmap which can be used to set 16 different general purpose events. After setting the respective GPE bit, it is checked if it was disabled by the guest. If not, an SCI is triggered to finally notify the guest.

```
void trigger_gpe(unsigned event_nr)
{
  // Activate this event in the appropriate register
  _gpe0_sts |=  0x00ff & (1 << event_nr);
  _gpe1_sts |= (0xff00 & (1 << event_nr)) >> 8;

  // If this event is masked by the guest, then just ignore it
  if ((0 == _gpe0_sts & _gpe0_en) || (0 == _gpe1_sts & _gpe1_en))
    return;

  // Send the guest an SCI
  MessageIrqLines msg(MessageIrq::ASSERT_IRQ, 9);
```

```
    _mb.bus_irqlines.send(msg);
}
```

Listing 4.6: ACPI Controller Method to Trigger a GPE

Triggering a hot plug event alone, which can mean that both a device is to be unplugged or it was replugged, is not sufficient, yet. To enable the guest OS to inform itself in its ACPI event handler about what exactly happened, two new registers are added: `PCIU` and `PCID`. When a bit is set in `PCID` (*D* as in *Detach*), this tells the guest that the PCI slot at the offset of the bit has to be unplugged. `PCIU` (*U* as in *Update*) works analogously, but its bits represent freshly replugged PCI slots. To raise such an event, a new *ACPI event* bus was added. Other modules within the VMM can now send `MessageAcpiEvent` messages which denote which virtual PCI slot has to be replugged or unplugged.

Additionally, a third new register is added. `B0EJ` acts as a write-clear register. Whenever the guest writes a bit in it, the VMM interprets this as "driver unloaded, please power off this slot", with the bit position indicating the slot number.

DSDT part

ACPI provides an elegant method to make guest systems react correctly to hardware-defined events: the *Differentiated System Description Table* (DSDT). DSDT entries contain *ACPI Markup Language* (AML) blocks of arbitrary size which are compiled from hand written *ACPI Source Language* (ASL) code. ASL code is compiled to AML using the Intel iASL compiler [18]. After compiling, the code is represented as a static C string within a C header file, which can easily be included into the project. Code copying the whole AML block string into the DSDT is embedded into the virtual BIOS restart procedure which also initializes all the other ACPI tables.

Code Listing A.1 on page 65 shows the ASL source code of the whole AML block in use. The `Scope(\_SB) {...}` block takes about two thirds of ASL code. It contains legacy PCI IRQ routing information as well as information about bus enumeration, usable I/O ports and memory ranges for mapping PCI *Base Address Registers* (BAR). Linux usually autodiscovers these values at boot. For hot plugging they are of importance again, because the hot plugging driver of Linux, which reconfigures new PCI devices after plug in, does not do the same.

Of direct relevance for the hot plugging workflow is the `Scope(\_SB.PCI0) {...}` block beginning at Line 230. For every PCI slot, it defines a `Device (Sxx) {...}` block (with `xx` being the slot number) which defines address, PCI slot number and an eject callback procedure. Furthermore, it defines three procedures:

`PCNF()` This method is called by the GPE handler triggered by the VMM. It reads the `PCIU` and `PCID` registers and calls the `PCNT()` procedure for every bit which is set on the according register.

`PCNT(device, event)` Dispatching by the `device` variable, this method calls `Notify(Sxx, event)` on the corresponding `Sxx` PCI slot object. `event` can have the value `1` which stands for "device was plugged in" and `3` which means "device eject requested". In reaction to the `Notify` call, the appropriate handler routine within the Linux kernel is started.

`PCEJ(device)` The `_EJ0` callback function of every device calls this method with its own device slot number. It writes the value `1 << device` to the `B0EJ` register. The VMM then knows that the device unplug is complete.

The last code block, `Scope (\_GPE) {...}` binds GPE bit 1 to a `PCNF()` call.

Hot plugging workflow

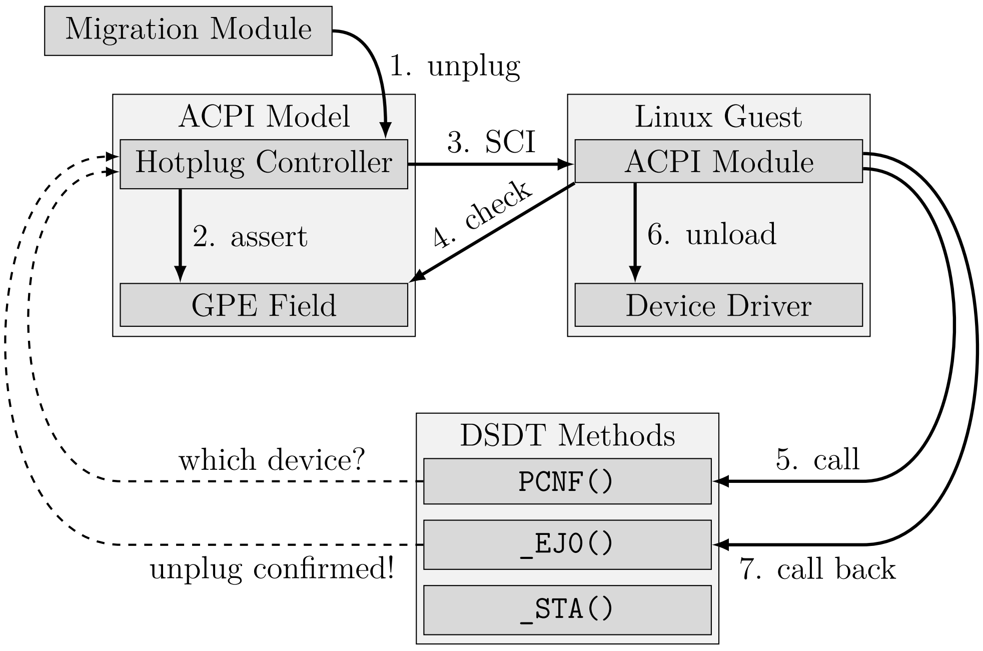

Figure 4.5.: High Level Overview of the ACPI Unplug Procedure

Using the example of the *unplug* event, the interaction between ACPI controller model and guest system shall be illustrated. See also Figure 4.5:

1. The unplug event can simply be triggered by an ACPI event message from within the VMM:

```
MessageAcpiEvent msg(MessageAcpiEvent::ACPI_EVENT_HOT_UNPLUG,
    slot_number);
_mb.bus_acpi_event.send(msg);
```

2. After receiving the message, the ACPI controller model sets the corresponding bit in its `PCID` register and triggers GPE 1.

3. The guest OS receives an interrupt and executes its SCI handler, which looks into the PM and GPE registers to determine the event that occured.
4. In reaction to the GPE bit which was set, the `_E01` handler method from within the AML block in the DSDT is called, which in turn calls `PCNF()`.
5. The `PCNF()` method uses the `PCIU` and `PCID` registers to distinguish which devices need to be unplugged or were replugged. It then calls `PCNT()` on the devices with the respective event type, which sends a notification to the guest OS.
6. Linux as the guest OS receives the notification in form of a notification handler call, which it hooked into its internal ACPI tree before. In reaction to this call, the device is unloaded. The BARs of the PCI device are unmapped and interrupt routes freed.
7. To tell the mainboard that it can power the PCI device off, Linux will call the `_EJ0()` callback method of the device slot, which in turn calls `PCEJ(<slot_nr>)`. `PCEJ()` writes the same bit which was set in the `PCID` register back to the `B0EJ` register. The VMM intercepts this write and marks the unplug event as processed.

Replugging works analogously, but is less complicated. The VMM sends a message with the `MessageAcpiEvent::ACPI_EVENT_HOT_REPLUG` parameter. Until Step 4, the workflow does not differ, but `PCNT()` will then be called with event type 1. The event type 1 handler for device hot plugging within the Linux kernel initializes the PCI slot and loads the device driver without notifying the VMM back. Consequently, as the guest reports back after unloading its device driver, unplugging a device can easily be implemented as a synchronous function. Due to the lack of reporting back by the guest, this is not as easy with plugging a device in. However, at this stage this is not a limitation.

4.3.2. Triggering Hot Plug Events During Live Migration

Being able to hot plug and unplug devices during VM runtime, this feature can finally be embedded into the existing live migration algorithm. The software architecture of the VMM makes integrating this feature very easy: For every PCI device which is mapped directly into the VM, an instance of the class `DirectPciDevice` is added to the set of device model objects within the VMM. This device model is not a *model* as such, since it does not model a device. It merely sets up the guest PCI configuration space, handles interrupt installation, DMA remapping, etc. However, being added to the VMM like any other device model, it can be hooked into the restore bus. When called by the restore bus during the migration procedure, it would not serialize and deserialize its state before/after host switch, like other device models. Instead, it would trigger its own hot unplug/replug event in the

ACPI controller. Code Listing 4.7 shows how the device identifies itself and sends the ACPI controller the respective hot plugging event. `msg.write` is *false* for device state retrieval before the host switch and *true* in the restore phase at the destination host.

```
bool receive(MessageRestore &msg)
{
  if (msg.devtype != MessageRestore::PCI_PLUG) return false;

  unsigned slot = (_guestbdf >> 3) & 0x1f;

  MessageAcpiEvent amsg(
    msg.write ?
      MessageAcpiEvent::ACPI_EVENT_HOT_REPLUG :
      MessageAcpiEvent::ACPI_EVENT_HOT_UNPLUG,
    slot);

  _mb.bus_acpi_event.send(amsg);
  return true;
}
```

Listing 4.7: MessageRestore Handler of the Helper Model of Directly Mapped PCI Devices

The restore bus is used for device state collection during the freeze round on the source host. On the destination host it is used for writing back all device states shortly before resuming the VM. The point in time during which device states are written back is also perfect for hot replugging pass-through NICs. Hot unplugging them in the device state collection phase, however, is not possible this way, because this cannot be done with a frozen VM. As the VM has to participate in unplugging PCI devices, the unplug message has to be sent over the restore bus earlier.

Choosing the right moment for unplugging pass-through NICs before migration is challenging for different reasons. The first reason is that unplugging a device is not finished until the guest writes back success into the `B0EJ` register of the ACPI controller. Measurements on the development machine have shown that this takes about 200 ms. Depending on the number of devices which need to be unplugged and on the guest OS version, etc., this number might vary. Furthermore, it is problematic to issue the unplug event in the last memory resend round before guest freeze, since it is never clear if the current round is the last round. Another circumstance is the fact that the existing live migration solution does not support guest memory tracking to detect page dirtying by DMA. As the pass-through NIC uses DMA, memory changes introduced by it during live migration might possibly stay unrefreshed on the destination host.

The actual implementation unplugs pass-through devices right before sending the first guest state byte at the beginning of the migration procedure. This solution both circumvents the missing DMA page tracking and effectively lowers the WWS for VMs with network-heavy workloads.

4.4. Guest Configuration with ACPI and Bonded NICs

The VMM now supports unplugging PCI devices, using the ACPI interface commodity operating systems support. To support general PCI hot plugging, the Linux kernel does not need any patches. It merely needs to be reconfigured if the VM is not using a stock kernel from any distribution, where this feature is usually already activated. This section describes how the guest system used for this project was changed. The distribution in use is the highly configurable *Buildroot*[2]. Buildroot is configured to produce a Linux kernel binary together with a complete root file system image.

Listing 4.8 shows the menu points of Linux's `make menuconfig` configuration menu, which have to be activated to support ACPI PCI hot plugging.

```
-> Power management and ACPI options
  - ACPI Support [y]
-> Bus options (PCI etc.)
  -> Support for PCI Hotplug [y]
    - ACPI PCI Hotplug driver [y]
```

Listing 4.8: Linux Menuconfig Entries for ACPI Hot Plugging of PCI Devices

The Linux kernel will recognize the hot plug ability of the virtual system at boot, and load and initialize its hot plug driver. A VM configured like this smoothly survives live migration. However, this does not account for network connections of user space applications utilizing the fast pass-through NIC. If it is unplugged while in use, the guest kernel has to inevitably kill all connections which depend on it. This scenario is similar to the use case of servers being equipped with multiple NICs for redundancy. Linux provides the *bonding* driver for this purpose, which was introduced by Donald Becker in Linux version 2.0 and is still being actively maintained.

With the bonding driver, the administrator can add *logical* NICs to the system. Having added such a logical NIC, multiple NICs can be *enslaved* to it. Figure 4.6 shows the internal network topology in a high level overview of all software stacks. NIC A is multiplexed by the host for use by host apps as well as virtual NIC models of VMs. To the Linux guest, it is visible as `eth0`. NIC B is directly passed through to the VM, visible by the guest system as `eth1`. Both are enslaved to the logical NIC `bond0`, which acts as the NIC, promised to be always logically connected to the network. The `bond0` device can be configured to combine its backend NICs in various ways: One obvious possibility is to aggregate all NICs to make them work together to sum up their throughput-rates. This can be done in different flavors, like static or dynamic load balancing, etc. The other possibility is the *active-backup* setting, where only one NIC is used at a time. Whenever this NIC fails, the bonding

[2]Buildroot is a set of makefiles and scripts which can be used to configure and compile compact GNU/Linux systems. `http://buildroot.uclibc.org`

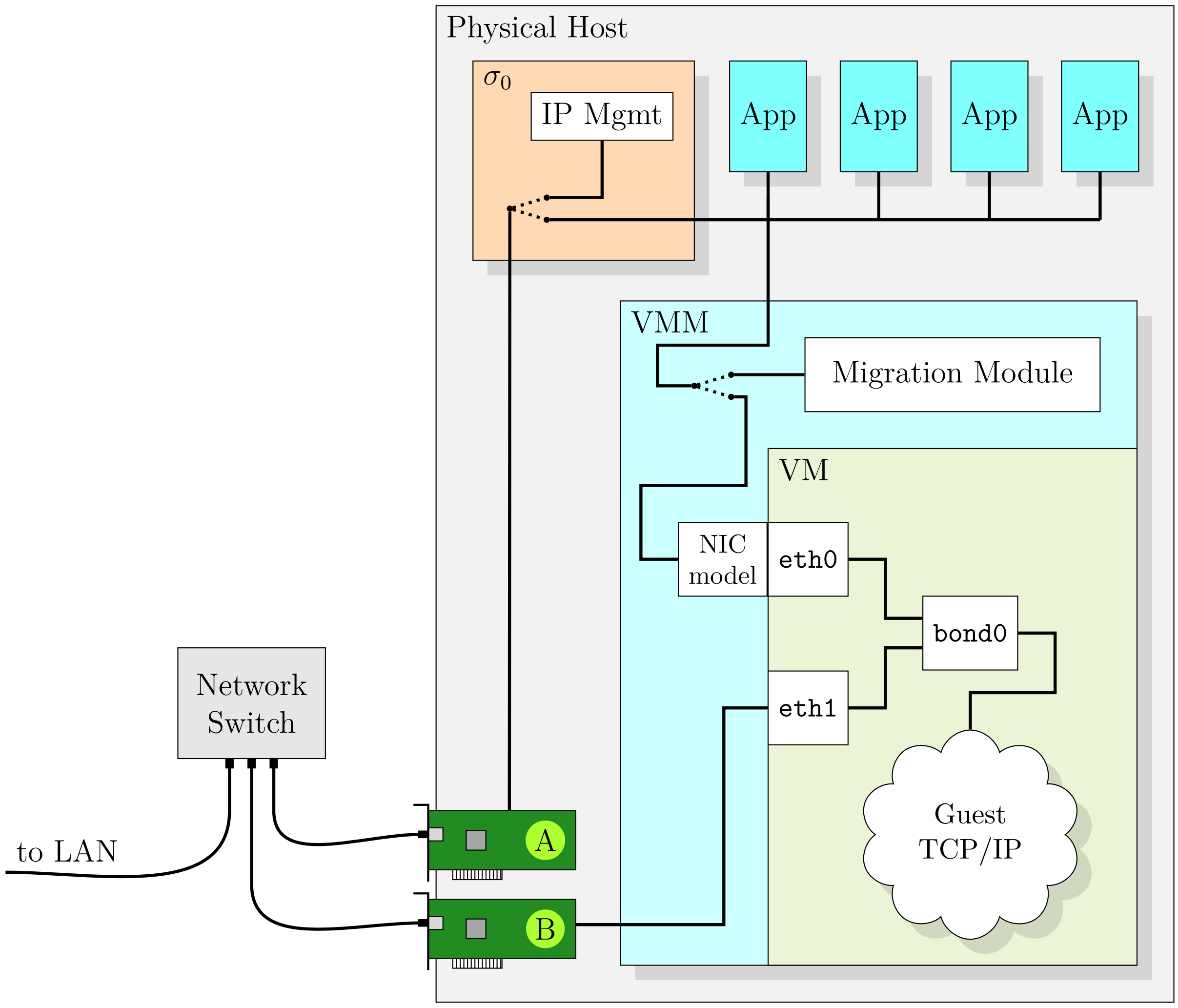

Figure 4.6.: Network Topology Inside the Virtualization Environment

driver will select another NIC from the set of enslaved devices to be the one being actively used. To advertise the NIC change to the network, the bonding driver automatically sends out gratuitous ARP packets from the active device. This is obviously the right choice for the scenario at hand. [8]

4.4.1. Configuring the Bonding Driver

To make the guest system operate the bonding driver in active-backup mode, the guest kernel has to be recompiled with bonding module support. During system boot, scripts need to bring up the bonding device into the respective state. Live migration introduces a third necessity when hot plugging a new NIC into the guest system, as the bonding driver does no automatic enslaving of new devices. An automatically executed script needs to do this job.

Kernel Modules

The bonding driver is another Linux kernel module which needs to be activated in the kernel. Listing 4.9 shows the menu entry which has to be activated in Linux's menuconfig configuration menu. The driver can be built as a module named `bonding`, or embedded into the kernel binary.

```
-> Device Drivers
    -> Network device support
        -> Network core driver support [y]
            - Bonding driver support [y]
```

Listing 4.9: Linux Menuconfig Entries for the Bonding Driver

After recompile and boot of the kernel, the system will be able to provide bonding devices, but they still need to be configured. To enslave NICs to a network bond, another tool called `ifenslave` was introduced with the bonding driver. Its source code resides in the `Documentation/networking/` directory within the Linux kernel source code repository. Compiling is done via `gcc ifenslave.c -o ifenslave`. It should then be moved into a system folder for binaries to be accesible for use.

Configuration Files

Setting up network connections in a late stage of the system boot of a GNU/Linux OS is usually done by SystemV-scripts. In this case, `/etc/init.d/S40network` sets up the network connections of the guest system. To initialize the bonding device ahead of the network configuration script, another init script was added to the init folder.

Listing 4.10 shows this script. In Line 7, a variable `slaves` is set to contain a list of all NICs to be enslaved. In this case, all NICs beginning with the name `eth` will be enslaved. The `1`, which is echoed in Line 11 into the `mode` file of the virtual *sys* file system tree of the bonding device, stands for the *active-backup* mode. In Line 15, the `eth0` device is defined as *primary* bonding device. This is the pass-through NIC. The bonding device is then finally set up and devices are enslaved to it in Lines 21 and 22. When the network initialization script is executed after this, it can successfully bring networking up.

Another configuration step which has proven to be cricital resides in Line 19 of the script. The *Media Independent Interface* (MII) monitoring frequency parameter is set to 100 ms. Hot unplugging of the active NIC alone does not raise any kind of event within the bonding driver. Without MII monitoring the network connection would just appear dead without any consequences. Thus the MII monitor has to be set to some polling frequency. The bonding driver will then be alerted in the next polling interval when the NIC appears dead and eventually activate the backup NIC.

```
1 #!/bin/sh
2 # file: /etc/init.d/S30bonding
3
```

```
case "$1" in
    start)
        echo "Configuring bonding device..."
        slaves=$(/sbin/ifconfig -a | \
            awk '/^eth/{gsub(":","",$1); print $1;}')

        # Set bonding mode to active-backup
        echo 1 > /sys/class/net/bond0/bonding/mode

        # Primary slave (the real NIC) which should be
        # used whenever possible
        echo eth0 > /sys/class/net/bond0/bonding/primary

        # Check for interface link every 100 ms. If not
        # set, active_slave is never switched if link is lost
        echo 100 > /sys/class/net/bond0/bonding/miimon

        /sbin/ifconfig bond0 up
        /sbin/ifenslave bond0 $slaves
        ;;
    stop)
        echo -n "Removing bonding device..."
        /sbin/ifconfig bond0 down
        ;;
    restart|reload)
        "$0" stop
        "$0" start
        ;;
    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
esac

exit $?
```

Listing 4.10: SystemV Init Script Setting Up the NIC Bond

On GNU/Linux distributions which follow the Debian style of network configuration, the network setup looks even simpler. In this case, the `/etc/network/interfaces` file has to be adapted as shown in Listing 4.11.

```
# file: /etc/network/interfaces

# ... loopback, etc. left out

auto eth0
iface eth0 inet manual
    bond-master bond0

auto eth1
iface eth1 inet manual
    bond-master bond0

```

```
auto bond0
iface bond0 inet dhcp
    bond-slaves eth0 eth1
    bond-primary eth0 eth1
    bond-mode active-backup
    bond-miimon 100
```

Listing 4.11: The Interfaces File as it Would Look Like on Debian Derivatives

Being configured this way, the booted Linux system will successfully bring up the bonding device, enslave `eth0` (pass-through NIC) as primary and `eth1` (virtual NIC) as secondary backend NICs. `eth1` will stay inactive until `eth0` fails or is unplugged.

Re-Bonding of Hot Plugged Devices

When deployed to practical live migration use, the current configuration will fail at the point at which a new pass-through NIC is hot plugged into the VM on the destination host. Neither the hot plug driver, nor the bonding driver automatically reenslave NICs to the bond. After unplugging `eth0`, the pool of NICs the `bond0` device can switch to is reduced to the virtual NIC. The freshly replugged `eth0`, which is a completely new NIC to the system, needs to be added to the bond.

Linux provides *event hooks* which can be used to launch scripts reacting to certain events. The *udev* utility, which is already used to populate the `/dev/` folder at boot, can be exploited versatilely for all kinds of device manipulation. It handles all user space actions when adding or removing devices, including firmware load. To make use of udev at this point, it is necessary to write a *udev rule*. Upon any unplug, replug, driver load/unload event, udev will look for matching rules in rules files in the `/etc/udev/rules.d` folder. If a rule applies, all actions attached to it are executed. Listing 4.12 shows the necessary rule for live migration. The event the rule has to match is the *add* event, which occurs after hot plugging a new device into the running system. The `SUBSYSTEM` specifier tells that this rule shall only apply to add-events which concern networking devices. After filtering out all other irrelevant events, udev would then launch the command specified in `RUN`. In this case, it will start the reenslave script, equipped with environment variables like `$INTERFACE`, carrying additional information. The reenslave script in Listing 4.13 makes sure that only devices with a name beginning with `eth` are enslaved to the bond.

```
# file: /etc/udev/rules.d/10-local.rules

ACTION=="add", SUBSYSTEM=="net", RUN="/bin/sh /bin/reenslave.sh"
```

Listing 4.12: Udev Rule Which Matches to All Freshly Plugged in NICs

```
#!/bin/sh
# file: /bin/reenslave.sh

if [[ "$INTERFACE" =~ ^eth ]]; then
    /sbin/ifenslave bond0 $INTERFACE
```

```
fi
```

Listing 4.13: The Reenslave Bash Script

Based on the design decisions from the previous chapter, this chapter has shown how to implement native ACPI hot plugging for guest systems of the underlying VMM. The existing live migration feature has been extended to support hot plugging pass-through devices during live migration. As it could be left largely unmodified, it has proven to be easily extensible. Guest systems with pass-through NICs do not automatically make use of the new possibility for network connection preserving. However, it has been shown using the example of GNU/Linux that they can easily be configured to make use of the new feature and do not need any code changes. Furthermore, the solution at hand is very robust and versatile as it does automatically handle the cases of different pass-through NIC models or missing pass-through NICs at the destination host.

5. Evaluation

Most important performance numbers of live migration algorithms are the *overall migration duration*, *service degradation* during the migration process, and the *total downtime* of the VM during host switch at the end of its migration. All these numbers can be measured by two experiments. The next two sections provide measurements of the network *latency* and *throughput* of a virtual machine during live migration. Both experiments were carried out with a VM running i686 Linux with kernel version 3.10.0-rc4. The VM was configured with 128 MB memory and two NICs: a virtual model of a virtual function device of the Intel 82576 network controller and a physical pass-through PCI card with the same controller model. Both measurements were performed with the packet MTU set to 1500.

5.1. Network Latency During Migration

A small application named *pingpong* was developed to do the same like usual ping pong tests of all kinds of communication libraries. The application can both act as network client and network server. It is started as a network server on the VM, listening on a TCP socket. On another host in the same network, it is started as a client and connects to the server listening on the VM. The client does then send a few bytes over TCP and the server will immediately echo them back. After receiving its message back, the client prints out the time it took for sending and receiving again. This is done in a loop.

The experiment flow is as follows: A VM is started on host A. Host B in the same network is ready to accept a migration request from the VM running on host A. Pingpong is started as a server on the VM. On a third host, pingpong is connected as client to the VM pingpong service. Its output is logged to a file for later processing. The next step is the initialization of the live migration procedure between host A and B. After the migration has succeeded, the pingpong measurement is stopped. While this is a common way to measure live migration algorithm performance, the specialty of this scenario is the VM configuration: Host A and B both give the VM access to a pass-through NIC.

Figure 5.1 shows the expected output graph of a fictional live migration with pass-through devices. The pingpong application measurement starts at second 0. This point in time lies somewhere in the lifetime of the VM. About 20 seconds after starting the pingpong measurement, the actual migration is initiated. At this point the pass-through NIC is hot unplugged and the service latency goes up, because the virtual NIC is slower. Switching between the real and the virtual NIC should be done without any additional latency. At second 40, the migration is finished and host B finally continues the execution of the VM. Host switching again does not involve any additional latency in the ideal case. As the hot replug event is injected into the VM even before its execution is continued, the bonding driver should immediately make use of it. After migration, the pingpong latency then returns down to the level it was at before the migration.

The implementation quality can be judged from the following details:

real-virtual NIC switch The ideal solution shows no measurable overhead by showing higher latency during NIC switching. Although this does not seem realistic, the implementation should minimize this latency.

virtual-real NIC switch Switching back from the virtual to the real NIC again ideally involves no latency peak. The implementation should also minimize this.

Figure 5.2 shows the output graph of a real-life scenario. The general appearance of the graph matches the graph expected from theory. However, as already suspected, latencies occur when NICs are switched. The latency which results from switching between the real and the virtual NIC is relatively high. Hot unplugging the pass-through NIC makes the bonding driver within the Linux kernel switch to its virtual NIC. The VM is not reachable for 400 ms in this situation. Hot plugging the pass-through NIC on the destination host is done immediately before resuming VM execution, but it takes 3.65 s until the guest effectively uses the pass-through NIC. Fortunately, the actual switch between virtual and real NIC does not add any measurable latency.

During host switch the downtime of the VM is also very high. It is not reachable for 1.7 s. This number is not to be accounted to any of the pass-through migration code changes. Prior versions of the migration feature, which were not able to migrate NICs of any kind, usually involved downtimes in the order of hundreds of milliseconds during host switch. The main part of additional downtime hence comes from introducing the virtual NIC model to the restore procedure. Especially time consuming here is the transfer of the relatively large packet queue buffers. Another cause is the time it takes all relevant hosts and switches to receive and process the gratuitous ARP packet sent out by the destination host to advertise the new physical location of the VM.

5.2. Network Throughput During Migration

To measure the effective network throughput of a VM while it is live migrated, a very similar experiment compared to the prior one was performed. The only difference is the measuring application. Instead of running pingpong, another program to measure the network throughput was written. The instance running on the VM acts as a server and continuously streams data to the client which periodically contains the current host CPU load.

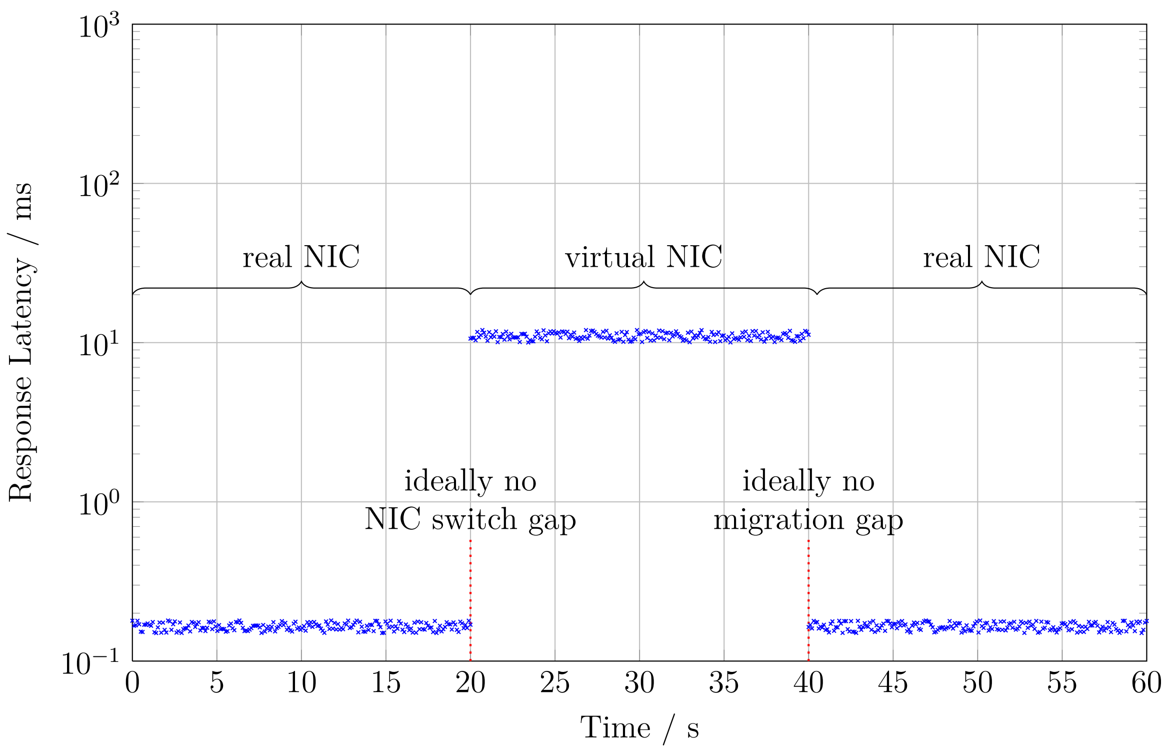

Figure 5.1.: Expected Network Latency of a VM During a Fictional Live Migration

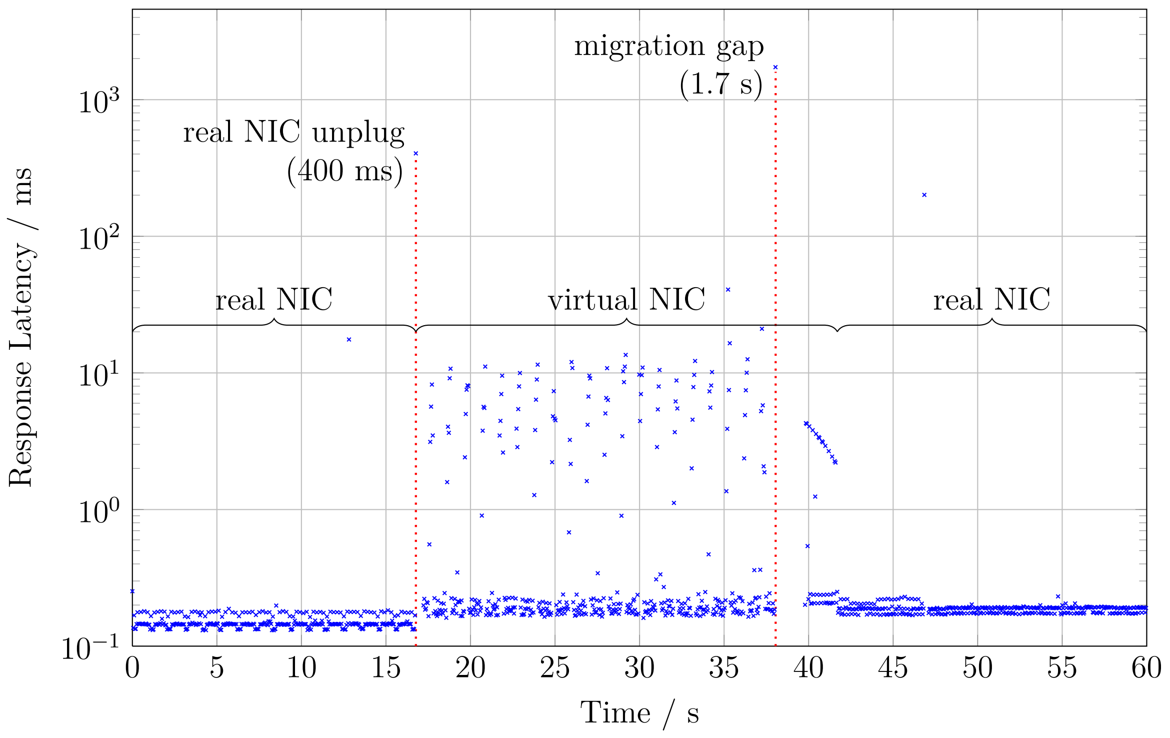

Figure 5.2.: Measured Network Latency of a VM During the Live Migration Process

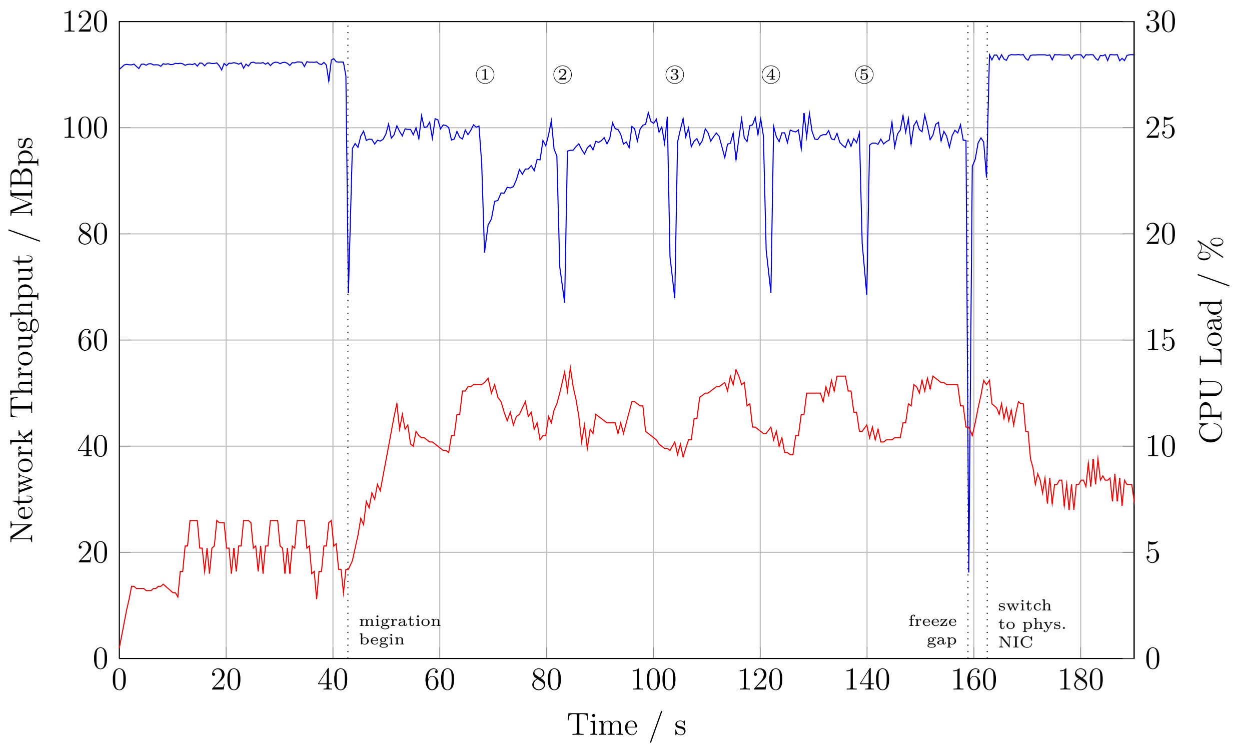

Figure 5.3.: Network Throughput of a VM (Upper Graph) and CPU Load of the Host (Lower Graph) During Live Migration

Figure 5.3 shows the result of the throughput measurement. The upper graph represents the network throughput of the VM during its own live migration. Comparing it to Figure 2.6 on page 13, it looks as expected from theory. The lower graph shows the total CPU load of the host system. It refers to the host the VM is currently being executed on.

At second 43, the live migration procedure is started. The fast pass-through NIC is unplugged, which immediately causes the network throughput to drop. Guest system and host now both use the same physical NIC, hence they have to share its maximum throughput-rate. The host does not run any other applications causing relevant CPU load or network traffic. Simultaneously, the CPU load of the host machine rises because the virtual NIC model of the VM must be serviced by the VMM in software. The overhead regarding CPU load when using a virtual NIC model adds up to about 5 %. At second 159 the throughput drops dramatically during the downtime of the VM while it is frozen shortly before the host switch. The VM does immediately continue to send data over the virtual NIC of the destination VMM. Although the pass-through NIC of the destination system is immediately hot plugged into the VM after it arrived, it takes the guest Linux system additional 3 seconds to actually use it. This is the time the guest system needs to initialize the device, load the appropriate driver and make the bonding device switch to it as its new active backend NIC. At second 162, the physical NIC is activated by the guest and the network throughput rises to its old maximum again.

Not directly relevant to the thesis project, but also interesting, are the short throughput drops which are numbered from ① to ⑤. From migration begin at second 43 to ①, the guest memory range is initially sent over network. Then, from ① to ②, all changes the guest system did to its memory are resent. In total, there are 5 *memory resend rounds*. At the end of every (re)send round, the throughput drops because the packet queues run empty before they are filled for the next round. Also visible is the additional CPU load introduced by the migration code when it enqueues all dirty pages for transfer: For every resend round, the migration algorithm takes about 3 % CPU load for several seconds.

6. Conclusion and Future Work

Until recently, a consequence of using pass-through devices in virtual machines was that they were not migratable any longer. Research projects from the near past have shown that this can be circumvented. This thesis project demonstrates that extending existing VMMs providing live migration with automatic management of pass-through devices can be done with minimal administrative effort for the user. The VMM is now able to live migrate virtual machines equipped with pass-through NICs of any model enabling for maximum network throughput possible. Furthermore, network connections of services running on the VM in question can be preserved in their functional state during migration. This is shown to be possible even in the scenario of using different pass-through NIC models before and after a live migration. The VMM device configuration the guest OS is initially booted with can be temporarily reduced. Limited hardware equipment of hosts to which the VM is migrated during its life time can therefore be tolerated. No mechanisms involving paravirtualization to synchronize host and guest had to be introduced to achieve this. Guest kernel code and driver modules remain absolutely unchanged. Merely interfaces which already exist on real unvirtualized hardware have been used. Any commodity operating system is able to deal with ACPI events behind which live migration events can be hidden.

Apart from unchanged guest source code, the feature of maintaining active TCP connections intact during host switch and pass-through NIC change requires additional guest system configuration. It is shown that the configuration requirements are minimal and trivial to cover by the VM administrator or user.

Being fully device type and model agnostic in regards of hot plug management, the solution presented proves to be both simple and versatile. Although ACPI hot plugging is generally applicable to any kind of PCI peripheral device, the feature of network connection preserving could easily be extended to cover other types of communication devices. The guest OS solely needs to provide an indirection layer in form of a logical device driver frontend, similar to the bonding driver which has been used for ethernet devices.

The measurement results from the last chapter show that forcing a VM to use only virtual devices by hot unplugging its pass-through devices does not dramatically reduce its reachability or performance. Regarding the effective downtime of the VM after the events of hot unplugging a NIC before the migration or the host switch at the end of it, certain potential not only for optimization of the hot plugging mechanism becomes obvious. Before giving an outlook on what could be achieved with the new VMM feature in the last section, a few possible approaches for optimization are presented in the next section.

6.1. Architectural Suggestions for VMMs

To optimize for performance of the live migration of VMs with possibly sophisticated configurations, a few changes might be implemented in future versions of different VMM solutions. This section gives an overview on possible optimizations.

6.1.1. Tracking Host Access to Guest Memory

When the VMM emulates privilege-sensitive processor instructions, it manipulates guest memory on behalf of the guest system. The same is done by device models. To access guest memory from within the address space of the VMM, guest-physical addresses need to be converted to host-virtual addresses first. Both the instruction emulator and the device models consult the virtual memory bus structure within the VMM to perform this translation. As this involves adding a simple offset to guest-physical addresses, code modules usually obtain this offset once and then calculate every guest memory address themselves.

This becomes a problem when every guest memory write access needs to be monitored, e. g. for live migration purposes. Tracking memory writes of the guest system is easily done by write-unmapping guest memory mappings, as shown before. However, VMM access to guest memory cannot be monitored using the same way and has to be done with additional mechanisms.

Instead of accessing the memory bus structure of the VMM only once for address offset receipt, it should be accessed for every guest memory write. This way the migration code module could hook itself into the memory bus as a listener and track access by every device model as well as the instruction emulator. This would lead to bulky code sending a message to the memory bus for every guest memory access. To prevent code unreadability, this might be eased by implementing some kind of intelligent pointer class type automatically consulting the memory bus when dereferenced. The same should happen on memory writes using `memcpy()`. If a device uses a whole lot of sparse guest memory accesses, it would be inefficient to communicate over the bus every time. A smart pointer could be used recording the addresses it pointed to. This could be able to automatically send an *invalidate page range* message in its destructor.

An important optimization in form of fine-grained guest freezing which would reduce the visible downtime of migrated VMs, were made possible if this was implemented. In general, a special networking thread of the VMM receives packets for virtual NICs of the guest, copies them into the guest memory space and maintains the receive queue states of the virtual NIC models. At this time, guest packet reception is simultaneously paused when the guest system is frozen. Packets from the network which are directed to the guest system are then dropped. Regarding TCP connections, this forces the sender to resend them as soon as the guest can receive packets again, which will be on the destination host. As the guest memory access tracking would now also detect host writes, the virtual NIC model freeze on the source host could be stalled until all other frozen devices states are sent to the destination host, together with dirty memory. Virtual NIC models together with the memory they dirtied by receiving guest packets for the frozen guest could then be sent to the destination host at last. This would reduce the number of dropped packets which arrived in the VM while it was frozen. The virtual NIC the VM would then continue working with on the destination host would consequently immediately give the guest OS access to as recent as possible packets. This would give it the chance to immediately acknowledge them over network, possibly even before the sender tries to resend them, resulting in a lower observed downtime.

6.1.2. Optimizing the Memory Resend Mechanism

At the current state, the migration algorithm sends a set of pages on every resend round and waits until everything has been successfully received by the other side. After this stage the algorithm checks which memory pages have been dirtied during that time. The set of dirty pages is the set of pages which is to be resent during the next round.

Further analysis of this set might reveal knowledge about how often which guest regions are actually used. The memory space could be polled for dirty pages multiple times per resend round. Doing this might enable maintaining an approximation of the current WWS of the guest system. Knowing the approximate size of the WWS would then in turn allow reasonable assumptions about how many resend rounds remain until VM freeze. PCI devices could then be unplugged nearly as late as possible.

6.1.3. Bonding Guest Driver Optimizations

Measuring the ACPI hot unplug procedure has revealed that, depending on the device and its driver, hot unplugging a NIC device takes time in the order of hundreds of milliseconds. Regaining network reachability for a pass-through NIC after replugging it can in turn take time in the order of seconds. These are time spans the VMM has no influence on. While a delay of hundreds of milliseconds to unplug a device might be acceptable, it seems reasonable to invest work on the reduction of the time it takes the guest system to make use of a freshly replugged device, which is in the order of seconds.

When the guest kernel receives the SCI interrupt indicating a device replug and handles it with its ACPI subsystem module, it starts a chain of asynchronous operations. First, the new PCI device needs to be indentified. Afterwards, the appropriate driver needs to be loaded which will then set it up by registering new device structures and interrupt handlers in the kernel and initializing the device itself. The initialized device can then be used by the bonding driver. It will set up the network connection, send gratuitous ARP packets and finally deactivate the other NIC, which was the virtual one.

The time it takes *after* loading the driver and making the new NIC go online, before all network participants have updated their routing tables, is not reducible.

The overhead of loading new drivers, however, can be mitigated when assuming the possibility that the destination host of the migration might give the guest a pass-through device of the same model as before. Instead of completely unloading device drivers, the guest system could just *mark* driver structures of unplugged NICs as *deactivated*. In the event that another NIC of the exact same type will be replugged, the kernel would then only need to execute a minimal amount of code to initialize its state without a complete driver reinitialization.

Another problem which might even be considered being a bug in the Linux kernel, is the fact that an ACPI hot unplug event alone does not trigger the bonding driver to switch NICs. Hot unplug events should immediately reach the bonding driver when they occur. This way it would become unnecessary to use the ARP polling or MII monitoring features to detect the unplug of the active NIC. Due to their polling nature they react later than an event-based solution would.

6.1.4. General NIC Hardware Improvements

An optimization which could come along with the idea of fine-grained guest freezing would be to enable NIC hardware to send acknowledgements for TCP packet sequences. The model of such a NIC within the VMM would then do the same, even when the VM is frozen. Network clients which send data to the VM would then not have to wait for the freshly resumed VM to acknowledge packets it should have received before the host switch phase of the migration. They could just continue to send the VM new packets as the source host has acknowledged the old ones already.

Another improvement of different nature would be a hardware serialization/deserialization feature. If hardware device states would be serializable and deserializable, the use of hot plugging during live migration would not be necessary, if pass-through devices of the same model are available on both participating hosts. Serialization and deserialization could be implemented by hardware designers in form of special I/O commands. Software might just send the device the command to serialize/deserialize, paired with an address range. The device would then write/read its state to/from this address. Complicated efforts as in the work of Pan et al. would then be obsolete [29].

However, this would still not be of great help if the destination host cannot provide the same pass-through device model after migration. If it was possible to generalize the format which is used by devices to describe their state, the deserialization hardware routine of a device model B could be fed with the serialization output of a device model A. In the case that the destination host cannot provide a pass-through device at all, a virtual model could be initialized with the generalized state information. Effects of live migration introduced by the use of hot plugging, like reduced guest network throughput and increased host CPU load during migration, would then disappear. Furthermore, guest systems would not need to be configured for this situation. Use of the bonding driver could be waived, at least for migration purposes.

6.2. Outlook

Hot plugging has proven to add much flexibility to VMs without the need to abandon faithful virtualization and at negligible costs. Hot plugging can also be done with CPUs and memory. When migrating a VM which makes use of e. g. 16 CPU cores on host A to a host B which can only provide 8 cores, the difference of cores could be unplugged. Running 8 virtual CPUs on 8 real cores might be more efficient on some platforms. Furthermore, if a VM was reduced to only few cores during live migration, the page dirtying rate could be reduced. Hot plugging of memory could be used for the same purpose. If the amount of memory of a VM is reduced prior to its migration, it would be forced to swap out parts of its memory onto its hard disk. If this is mounted over network from a shared drive, it would effectively reduce the amount of memory which needs to be sent and synchronized during the migration. With this set of features, the VMM could automatically compensate varying host resources for every VM regarding PCI devices, host CPUs and host memory, by using native ACPI interfaces.

If guest systems were designed for the scenario of sporadically disappearing PCI devices, CPUs, and memory, new use cases of virtualization could emerge. Regarding desktop computers, servers, cell phones and embedded systems like e. g. car computers as systems with similar, but heterogeneous architectures, the same guest system could be used on all of them. VMs could just *follow* the user, automatically hopping from host to host using live migration. Instead of having one user account on every system, users could move their user account in form of a whole VM between the systems they use. Users would then migrate their system e. g. from their workstation onto their cell phone and then carry it home.

Adopting the idea of meta drivers like the bonding driver in Linux used for networking, it might seem useful to unplug and replug GPUs, monitors, input devices, etc. from/to VMs while they *roam* between different hosts. Specialized user applications could then get the maximum out of currently available devices with flexible desktop environments and multimedia applications.

The next step on top of this vision would be to execute the virtual CPUs of single VMs on multiple hosts. Software would then be ultimatively decoupled from hardware. While users in the past had to configure and maintain all their computer devices individually, these would then be logically melted into a single, dynamically extensible system composition. This would again create a variety of use cases. Users of cheap laptop computers could for example *rent* computation time of a dozen additional virtual CPUs for their system from cloud providers to be able to execute resource hungry applications. Cell phones, tablets and desktop computers could be combined at home to provide more computation power. It might also be useful to start some kind of workload on a local VM and then send it into a cloud to collect the results of the workload later.

A. Code Listings

```
DefinitionBlock (
    "dsdt.aml", // Output Filename
    "DSDT",     // Signature
    0x00,       // DSDT Compliance Revision
    "BAMM",     // OEMID
    "JONGE",    // TABLE ID
    0x1         // OEM Revision
    )
{
    Scope(\_SB) {
        Device(PCI0) {
            Name(_HID, EisaId("PNP0A03")) // PCI Host Bridge
            Name(_ADR, 0)
            Name(_UID, 0)

            // Hot Plug Parameters. Optional.
            // Linux will complain and use standard
            // parameters, if not provided.
            Name(_HPP, Package() {
                0x08,  // Cache line size in dwords
                0x40,  // Latency timer in PCI clocks
                0x01,  // Enable SERR line
                0x00   // Enable PERR line
            })

            // PCI Routing Table
            // When defining as much ACPI information as
            // needed for hot plug, we also have to define
            // Interrupt routing tables like the following.
            // Otherwise, Linux would complain.
            Name(_PRT, Package() {
                Package() { 0x1ffff, 0, LNKA, 0 },
                Package() { 0x1ffff, 1, LNKB, 0 },
                Package() { 0x1ffff, 2, LNKC, 0 },
                Package() { 0x1ffff, 3, LNKD, 0 },

                Package() { 0x2ffff, 0, LNKA, 0 },
                Package() { 0x2ffff, 1, LNKB, 0 },
                Package() { 0x2ffff, 2, LNKC, 0 },
                Package() { 0x2ffff, 3, LNKD, 0 },

                Package() { 0x3ffff, 0, LNKA, 0 },
                Package() { 0x3ffff, 1, LNKB, 0 },
                Package() { 0x3ffff, 2, LNKC, 0 },
                Package() { 0x3ffff, 3, LNKD, 0 },

                Package() { 0x4ffff, 0, LNKA, 0 },
                Package() { 0x4ffff, 1, LNKB, 0 },
                Package() { 0x4ffff, 2, LNKC, 0 },
                Package() { 0x4ffff, 3, LNKD, 0 },
            })

            // At boot, Linux will either scan the system for
            // possible resources used by PCI cards or read
            // ACPI tables to obtain this information.
            // When providing as much ACPI data as needed
            // for hot plugging, then this is not optional any
            // longer. Linux would complain if all this was
            // not provided here.
            Name (_CRS, ResourceTemplate () {
                // Bus enumeration from _MIN to _MAX
                WordBusNumber (
                    ResourceProducer,
                    MinFixed,     // _MIF
                    MaxFixed,     // _MAF
                    ,
                    0x00,         // _GRA
                    0x00,         // _MIN
                    0xFF,         // _MAX
                    0x00,         // _TRA
                    0x100)        // _LEN
                // IO ports usable by PCI from _MIN to _MAX
                WordIO (
                    ResourceProducer,
                    MinFixed,     // _MIF
                    MaxFixed,     // _MAF
                    PosDecode,
                    EntireRange,
                    0x0000,       // _GRA
                    0x0000,       // _MIN
                    0x7FFF,       // _MAX
                    0x00,         // _TRA
                    0x8000)       // _LEN
                // System memory for mapping BAR
                // areas from _MIN to _MAX
                // BAR = Base Address Register,
                // every PCI card will
                // usually have 2 of those.
                DWordMemory (
                    ResourceProducer,
                    PosDecode,
                    MinFixed,     // _MIF
                    MaxFixed,     // _MAF
                    NonCacheable, // _MEM
                    ReadWrite,    // _RW
                    0x00000000,   // _GRA
                0xE0000000,   // _MIN
                0xE0FFFFFF,   // _MAX
                0x00,         // _TRA
                0x01000000)   // _LEN
        })

        // This block introduces three named dword
        // fields in IO space. The hot plug
        // controller implements these as virtual
        // I/O registers. During hot plug/unplug,
        // guest and the hosts hot plug controller
        // will communicate over these.
        OperationRegion(PCST, SystemIO, 0xae00, 12)
        Field (PCST, DWordAcc, NoLock, WriteAsZeros)
        {
            PCIU, 32, // IO port 0xae00
            PCID, 32, // IO port 0xae04
            B0EJ, 32, // IO port 0xae08
        }

        // Status method. Statically returns
        // "Everything is up and working"
        // because the PCI root bus will always be there.
        Method (_STA, 0) { Return (0xf) }
    }

    // All this interrupt routing information is necessary.
    // This defines the interrupts A, B, C, D, considered
    // legacy nowadays.
    // Hot plugging etc. will work without this anyway if
    // the PCI device uses MSI for interrupting, but the
    // kernel would complain with ugly error messages.
    // These device definitions are kept as minimal as
    // possible.
    Device(LNKA){
            Name(_HID, EISAID("PNP0C0F")) // PCI interrupt link
            Name(_UID, 1)
            Method (_STA, 0, NotSerialized)
            {
                Return (0x0B)
            }
            Method (_CRS, 0, NotSerialized)
            {
                Name (BUFF, ResourceTemplate () {
                    IRQ (Level, ActiveLow, Shared) {5}
                })
                Return (BUFF)
            }
            Method (_PRS, 0, NotSerialized)
            {
                Name (BUFF, ResourceTemplate () {
                    IRQ (Level, ActiveLow, Shared) {5,9,10}
                })
                Return (BUFF)
            }
            Method (_SRS, 1, NotSerialized) {}
            Method (_DIS, 0, NotSerialized) {}
        }
        Device(LNKB){
            Name(_HID, EISAID("PNP0C0F")) // PCI interrupt link
            Name(_UID, 2)
            Method (_STA, 0, NotSerialized)
            {
                Return (0x0B)
            }
            Method (_CRS, 0, NotSerialized)
            {
                Name (BUFF, ResourceTemplate () {
                    IRQ (Level, ActiveLow, Shared) {10}
                })
                Return (BUFF)
            }
            Method (_PRS, 0, NotSerialized)
            {
                Name (BUFF, ResourceTemplate () {
                    IRQ (Level, ActiveLow, Shared) {5,9,10}
                })
                Return (BUFF)
            }
            Method (_SRS, 1, NotSerialized) {}
            Method (_DIS, 0, NotSerialized) {}
        }
        Device(LNKC){
            Name(_HID, EISAID("PNP0C0F")) // PCI interrupt link
            Name(_UID, 3)
            Method (_STA, 0, NotSerialized)
            {
                Return (0x0B)
            }
            Method (_CRS, 0, NotSerialized)
            {
                Name (BUFF, ResourceTemplate () {
                    IRQ (Level, ActiveLow, Shared) {9}
                })
                Return (BUFF)
            }
            Method (_PRS, 0, NotSerialized)
            {
                Name (BUFF, ResourceTemplate () {
                    IRQ (Level, ActiveLow, Shared) {5,9,10}
                })
                Return (BUFF)
            }
            Method (_SRS, 1, NotSerialized) {}
```

```
                Method (_DIS, 0, NotSerialized) {}
        }
        Device(LNKD){
                Name(_HID, EISAID("PNP0C0F")) // PCI interrupt link
                Name(_UID, 4)
                Method (_STA, 0, NotSerialized)
                {
                    Return (0x0B)
                }
                Method (_CRS, 0, NotSerialized)
                {
                    Name (BUFF, ResourceTemplate () {
                        IRQ (Level, ActiveLow, Shared) {5}
                    })
                    Return (BUFF)
                }
                Method (_PRS, 0, NotSerialized)
                {
                    Name (BUFF, ResourceTemplate () {
                        IRQ (Level, ActiveLow, Shared) {5,9,10}
                    })
                    Return (BUFF)
                }
                Method (_SRS, 1, NotSerialized) {}
                Method (_DIS, 0, NotSerialized) {}
        }

    }

    Scope(\_SB.PCI0) {
        // These are PCI slot definitions.
        // They are necessary because every PCI card
        // which shall be ejectable, needs an _EJ0 method.
        Device (S01) {
           Name (_ADR, 0x10000)
           Name (_SUN, 0x01) // SUN: Slot User Number

           // This method is called by the operating system
           // after unloading the device driver etc.
           // _EJ0 = eject callback
           Method (_EJ0, 1) { PCEJ(0x01) }
        }

        Device (S02) {
           Name (_ADR, 0x20000)
           Name (_SUN, 0x02)
           Method (_EJ0, 1) { PCEJ(0x02) }
        }

        Device (S03) {
           Name (_ADR, 0x30000)
           Name (_SUN, 0x03)
```

```
            Method (_EJ0, 1) { PCEJ(0x03) }
        }

        Device (S04) {
            Name (_ADR, 0x40000)
            Name (_SUN, 0x04)
            Method (_EJ0, 1) { PCEJ(0x04) }
        }

        // Called by some PCI card's _EJ0 method,
        // This tells the VMM to turn off the
        // PCI device by writing (1 << PCI_ID) to the
        // IO port associated with the B0EJ symbol.
        Method (PCEJ, 1, NotSerialized) {
             Store(ShiftLeft(1, Arg0), B0EJ)
             Return (0x0)
        }

        // PCNT = PCi NoTify
        // PCNT(<device>, <1 = check for inserted device /
        //                 3 = eject requested>)
        // The values 1 and 3 are defined in the ACPI spec
        Method(PCNT, 2) {
             If (LEqual(Arg0, 0x01)) { Notify(S01, Arg1) }
             If (LEqual(Arg0, 0x02)) { Notify(S02, Arg1) }
             If (LEqual(Arg0, 0x03)) { Notify(S03, Arg1) }
             If (LEqual(Arg0, 0x04)) { Notify(S04, Arg1) }
        }

        /* PCI hot plug notify method */
        Method(PCNF, 0) {
             // Local0 = iterator
             Store (Zero, Local0)

             // These two fields contain bits mapped
             // to PCI devices, like in the GPE bitmap.

             // bit (1 << N) set here
             //   --> Device N was inserted
             Store (PCIU, Local1)
             // bit (1 << N) set here
             //   --> Device N has to be removed
             Store (PCID, Local2)

             While (LLess(Local0, 4)) {
                  Increment(Local0)
                  If (And(Local1, ShiftLeft(1, Local0))) {
                       PCNT(Local0, 1) // 1 => DEVICE CHECK
                  }
                  If (And(Local2, ShiftLeft(1, Local0))) {
                       PCNT(Local0, 3) // 3 => EJECT REQUEST
                  }
            }
            Return(One)
        }
    }

    Scope (\_GPE)
    {
        Name(_HID, "ACPI0006")

        // These methods are wired to the according
        // bits in the GPE bitmap. The VMM will
        // raise bits and then send an interrupt 9.
        // The ACPI code in the guest kernel will
        // then dispatch one of these methods.
        Method(_E01) {
            \_SB.PCI0.PCNF() // PCI hot plug event
        }
    }

} // end of definition block
```

Listing A.1: ASL Code of the DSDT AML Block Needed for Hot Plugging

B. Acronyms and Terminology

DHCP	The *Dynamic Host Configuration Protocol* is a centralized protocol used by new network clients to obtain a new IP address from a central address assignment instance.
DMA	*Direct Memory Access* is a hardware feature that allows to copy memory between system memory and devices without utilizing the CPU
ISA	The *Instruction Set Architecture* is the part of the computer architecture visible to software. This includes instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.
Kernel	Main component of a computer operating system managing the resources of the system for applications
Kernel Module	The Linux kernel provides the possibility to load code at runtime to extend its functionality
MMU	The *Memory Management Unit* translates virtual addresses to physical addresses using page tables the operating system provides.
MTU	The *Maximum Transmission Unit* is the maximum size of an unfragmented network packet a system can transfer over network.
NIC	The *Network Interface Controller*, also known as Network Interface Card, or LAN Adapter Connects a computer to a computer network.
Page	On systems implementing the concept of virtual memory the whole memory range is partitioned into blocks (then called *pages*) of fixed size
SMP	The term *Symmetric Multiprocessing* describes a class of computer architectures where multiple cores share the same main memory space and are controlled by a single operating system instance

Bibliography

[1] AMD. Amd64 virtualization codenamed “pacifica” technology, secure virtual machine architecture reference manual. Technical report, AMD, May 2005. Revision 3.01.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP ’03, pages 164–177, New York, NY, USA, 2003. ACM.

[3] M. Ben-yehuda, J. Mason, O. Krieger, J. Xenidis, L. V. Doorn, A. Mallick, and E. Wahlig. Utilizing iommus for virtualization in linux and xen. In *In Proceedings of the Linux Symposium*, 2006.

[4] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGPLAN Not.*, 43(3):26–35, Mar. 2008.

[5] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS ’01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI’05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[7] A. L. Cox. Optimizing network virtualization in xen. In *In Proceedings of the USENIX Annual Technical Conference*, pages 15–28, 2006.

[8] T. Davis. *Linux Ethernet Bonding Driver HOWTO*, Apr. 2011. `https://www.kernel.org/doc/Documentation/networking/bonding.txt`.

[9] J. Fisher-Ogden. Hardware support for efficient virtualization. 17:2008, 2006.

[10] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Harvard University, 1973.

[11] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. PCI Hot Plug Specification, June 2001. Revision 5.0.

[12] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced Configuration and Power Interface Specification, December 6, 2011. Revision 5.0.

[13] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[14] Intel. Intel virtualization technology specification for the ia-32 intel architecture. Technical report, Intel, Apr. 2005.

[15] Intel. Intel virtualization technology for directed i/o architecture specification. Technical report, Intel, Sept. 2007.

[16] Intel. Intel 82576 SR-IOV driver companion guide. Technical report, LAN Access Division, June 2009. Revision 1.00.

[17] Intel. ACPI Component Architecture user guide and programmer reference. Technical report, ACPICA, June 2013. Revision 5.16.

[18] Intel. iASL: Acpi source language optimizing compiler and disassembler user guide. Technical report, ACPICA, Jan. 2013. Revision 5.03.

[19] A. Kadav and M. M. Swift. Live Migration of Direct-Access Devices. *SIGOPS Oper. Syst. Rev.*, 43(3):95–104, July 2009.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium Volume 1*, Ottawa, Ontario, Canada, June 2007.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[22] P. Kutch. Pci-sig sr-iov primer: An introduction to sr-iov technology. *Application note*, pages 321211–002, 2011.

[23] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM.

[24] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, Dec. 1995.

[25] J. Liedtke. Towards real microkernels. 39(9):70–77, Sept. 1996.

[26] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 101–110, New York, NY, USA, 2009. ACM.

[27] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[28] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, Sept. 2000.

[29] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang. Compsc: live migration with pass-through devices. *SIGPLAN Not.*, 47(7):109–120, Mar. 2012.

[30] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

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

[32] J. E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, Burlington, MA, 2005.

[33] U. Steinberg. *NOVA Microhypervisor Interface Specification*, Feb. 2013. preliminary version.

[34] U. Steinberg and B. Kauer. NOVA: a Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[35] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36:195–209, December 2002.

[36] E. Zhai, G. D. Cummings, and Y. Dong. Live Migration with Pass-through Device for Linux VM. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, July 2008.