Karlsruher Institut für Technologie

Interconnect Adapter State Migration for Virtual HPC Clusters

Diplomarbeit
von

cand. inform. Simon Sturm

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuende Mitarbeiter:	Dipl.-Inform. Marius Hillenbrand
	Dr. Jan Stöß

Bearbeitungszeit: 19. Juni 2012 – 18. Januar 2013

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 18. 01. 2013

Deutsche Zusammenfassung

Live-Migration von virtuellen Maschinen wird als eines der mächtigsten Werkzeuge im Bereich der Systemvirtualisierung angesehen. Es kann insbesondere in Rechenzentren erhöhte Fehlertoleranz durch Ausfallsicherungssemantiken und dynamische Lastverteilung in einem Cluster ermöglichen.

High Performance Computing Infrastructure as a Service (HPC IaaS) legte in der letzten Zeit erheblich an Bedeutung zu. Es verspricht Elastizität und Flexibilität in den Bereich des High Performance Computing (HPC) zu bringen. Im Gegensatz zu traditionellen *Ethernet*-basierenden Cloud Rechenzentren, benutzen HPC Cluster typischerweise hochperformante Verbundnetzwerktechnologien mit intelligenten Netzwerkkarten, wie beispielsweise InfiniBand. Sie erreichen eine erstklassige Kommunikationsleistung durch aggressives Auslagern des Protokollhandlings, OS-bypass-Techniken und erweiterte Funktionen wie Remote DMA. Obwohl die neueste Generation solcher Karten bereits Hardwarefunktionen zur Verfügung stellen, die ihre Virtualisierung erleichtern (bspw. mit SR-IOV), ist die Unterstützung für transparente Live-Migration noch immer eine ungelöste Fragestellung der Forschung.

In dieser Diplomarbeit stellen wir ein neuartiges Design vor, welches gasttransparente Live-Migrationen von virtuellen Maschinen ermöglicht, denen eine virtuelle Instanz einer selbstvirtualisierenden Hardware direkt durchgereicht wurde. Wir nutzen dabei gezielt die Selbstvirtualisierungsfunktion der Hardware aus um die eigentliche Zustandsmigration der durchgereichten Instanz durchzuführen. Obwohl wir uns auf HPC IaaS fokussieren, kann unser Ansatz auch in anderen Szenarien verwendet werden. Eine prototypische Evaluation, die auf einem *Mellanox ConnectX-3 VPI InfiniBand* Adapter und dem Linux KVM Virtual Machine Monitor (VMM) basiert, beweist, dass sich dieses Design in aktuelle VMMs integrieren lässt.

Abstract

Live migration of virtual machines is considered as one of the most powerful tools available in the context of machine virtualization. Specifically in data centers, it can enable higher fault tolerance through fail-over semantics and better load-balancing within a cluster.

High performance computing infrastructure as a service (HPC IaaS) has gained substantial momentum in the recent past, as it promises to bring elasticity and flexibility to the area of high performance computing (HPC). In contrast to traditional *Ethernet*-based cloud data centers, however, HPC clusters typically use high-speed interconnects with intelligent network adapters, such as *InfiniBand*. They provide cutting-edge communication performance and use aggressive protocol offloading, OS-bypass techniques, and advanced features such as remote DMA. While the latest generation of such cards already provides hardware capabilities to ease their virtualization (with SR-IOV, for instance), support for transparent live migration of these self-virtualizing adapters is still an open question in virtualization research.

In this thesis, we propose a novel design enabling guest-transparent live migration of a virtual machine configured with a directly assigned virtual device instance of self-virtualized hardware. We take advantage of the self-virtualization features to perform the actual state migration of the assigned instance in software. Despite we focus on HPC IaaS, our approach can be also employed in other scenarios. A prototypical evaluation, based on *Mellanox's ConnectX-3 VPI InfiniBand* HCA and the Linux KVM virtual machine monitor (VMM), proofs that this design integrates with recent VMMs.

Contents

1 Introduction and Motivation

Virtual High Performance Computing (HPC) clusters, or also referred to as High Performance Computing Infrastructure as a Service (HPC IaaS), is an upcoming trend in the computer service industry. Renting a scalable virtual HPC cluster on-demand instead of operating and owning a physical cluster by oneself promises new possibilities and opportunities for users and providers. However, HPC IaaS combines two different developments in computer science with different and colliding key aspects: (1) Machine Virtualization and (2) HPC. Virtualization focuses on machine abstraction, thus providing the guest an abstract view of the machine on which it is actually running. In contrast to this, HPC benefits from using huge amounts of parallel computing and memory resources as directly as possible to achieve high performance. Typical HPC applications are computational fluid dynamics, seismic data analysis, online transaction processing, and real-time control systems, amongst others [14].

Machine virtualization has developed over time and its beginning was primarily marked by a publication of Popek and Goldberg in the early 1970s [61, 62]. They provided a set of formal requirements for a computer architecture to enable machine virtualization. During that time, *IBM* was one of the first companies that commercially published a virtual machine environment for the *IBM System/370* mainframe [18]. It allows customers to consolidate multiple guest systems on a single physical host.

Machine virtualization is the key technology enabling IT infrastructure as a cloud service. Advanced features, such as migration, sever the binding of a virtual machine (VM) to a particular physical host. With live migration, which is seen as one of the most powerful tools, a VM can be moved to another host almost seamlessly - even while it is currently running. It enables higher fault-tolerance and is a useful feature for dynamically balancing load within a datacenter.

Nowadays, virtualization is becoming increasingly relevant to the area of HPC. Since hardware virtualization features became more and more publicly available, for instance with *Intel's VT-x* and *AMD's AMD-V* technology, performance oriented virtualizing is possible: They enable running a VM at almost native speed since most of the guest instructions can run directly on the physical hardware. This way, virtualization provides scalability and flexibility for HPC, as shown in recent research [21, 24, 33].

Supercomputing requires high performance interconnect technologies, since such computers are typically built from a large number of computer nodes that are interconnected with each other. The interconnect network becomes a vital part, since supercomputers achieve their performance from extreme computing parallelism and huge memory resources. For this purpose, these high-speed interconnects, such as *InfiniBand*, provide cutting-edge communication performance by utilizing aggressive protocol offloading, OS-bypass technologies, and advanced features such as remote DMA (RDMA). With HPC IaaS, however, those technologies substantially complicate the task of transparently migrating a VM to another host. While the latest generation of such cards, such as the *Mellanox ConnectX-3 VPI InfiniBand* host channel adapter [42], already provides hardware self-virtualization to ease their use in virtualization scenarios with SR-IOV [59], for instance, transparent support for live migrating directly assigned devices is still an open question in virtualization research. Direct device assignment promises best I/O utilization results in virtualization [15] and migration transparency is the important key to provide full flexible cloud infrastructures services, because it removes the dependency to particular systems for the guest which support running on the according service.

In this thesis, we propose a novel design enabling guest-transparent live migration of a VM with a directly assigned virtual instance of self-virtualized hardware. For this purpose, we take advantage of the self-virtualization features of the hardware to perform the actual state migration of the assigned instance in software at the virtual machine monitor (VMM) layer. Despite we focus on HPC IaaS, we introduce our approach in a generic way, since we are confident that this approach is also a base for migrating states of other self-virtualized devices.

This thesis is structured as follows: We present some background information and analyze the challenges of device state migration in Chapter 2 on the facing page. We focus on self-virtualizing hardware in the area of infrastructure as a service (IaaS) and HPC. In Chapter 3 on page 27 we introduce our novel design. It consists of two orthogonal components that are composed together. The first component is a migration strategy and the second is a novel interface. Such a strategy coordinates the migration process in a way to enable migration transparency to the guest and also to remote nodes that are interacting through this device. The interface, that we call migration assistance interface (MAI), provides necessary mechanisms to a VMM to enable guest-transparent state migration of an assigned device. It utilizes the self-virtualization feature of the hardware to break up device state opaqueness. Additionally, we provide an *InfiniBand* specific slave device migration approach. In Chapter 4 on page 45, we evaluate our proposed design with focus on the integrability on Linux's KVM VMM stack. After that, we discuss related work in Chapter 5 on page 61 and contrast to our approach. Finally, in Chapter 6 on page 67, we conclude our work and summarize our results.

2 Background and Analysis

In this chapter, we provide some background and analyze challenges of live migrating virtual machines (VMs) to that a slave device of a self-virtualizing hardware is assigned. We focus thereby on the area of High Performance Computing (HPC), especially on the scope of HPC infrastructures as a cloud service. Since live migrating directly assigned devices covers a wide range of area, we present our analysis and background as generic as possible.

First, we provide relevant background on HPC as a cloud service in Section 2.1. After that, we point out how machine virtualization is utilized for HPC and what are the challenges for its implementation in Section 2.1.1 on page 7. In this section, we also analyze recent virtualization techniques, since machine virtualization covers resource virtualization of computing time, memory, I/O, and the high performance interconnect network. I/O virtualization is covered in more detail in Section 2.1.2 on page 10, because it imposes challenges for migration, especially to direct device assignment. After an introduction to self-virtualizing devices and an overview of their potential for HPC in Section 2.1.3 on page 12, we analyze the live migration process of virtual machines focused on virtual HPC clusters using self-virtualizing high performance interconnect adapters in Section 2.2 on page 15. Finally, we give a brief overview of *InfiniBand* and details of hardware-based *InfiniBand* adapter virtualization in Section 2.3 on page 20. Since we primarily focus our scenario on HPC clusters built with the modern *InfiniBand* interconnect, we shall introduce the concepts and properties of *InfiniBand*.

2.1 High Performance Computing in the Cloud

HPC in the cloud, also called HPC Infrastructure as a Service (HPC IaaS) [21], fuses HPC with the aspects of Cloud Computing. It delivers a flexible virtual HPC cluster as a service which promises more flexibility and reduced operating costs compared to a physical owned cluster. Furthermore, benefits of Cloud Computing are in general a better utilization, higher energy efficiency, and lower overall operation costs by consolidating multiple users on a single data center [13]. And because resources can be added or removed at a fine grain within short time periods, these resources can be matched to workload needs more closely [13].

Typical high performance applications are simulation, data mining, information access, or information integration [14]. Examples are computational fluid dynamics, seismic data analysis, online transaction processing, and real-time control systems [14]. One of the most noticeable properties of HPC workloads is the high degree of computation parallelism. This parallelism is exploited by many computation threads which are spread over multiple processors and even multiple HPC cluster nodes. A fast interconnect network is used to exchange interim results and for task synchronizations during processing (compare with Figure 2.1). However, the characteristics of high performance workloads differ significantly from typical server or workstation workloads in their high dependence on computational performance, data storage access rates, as well as communication transfer speed and low latency [14, 19].

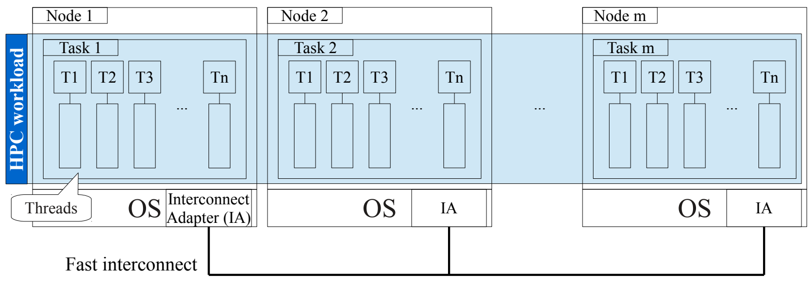

Figure 2.1: Typical runtime environment of an HPC workload

The term of Cloud Computing stands for a new trend of computing, the realization of the long-held dream of computing as a utility [13]. With new technologies, operators are now able to deliver services at different programming abstraction levels that can be quickly scaled on-demand and can be charged by usage [13]. Flexibility and illusion of infinity resources of the service are achieved by virtualizing the underlying resources and hiding the implementation details from the users [13]. By using cloud services, end-users do not have to operate the underlying infrastructure by themselves but rather can rent the needed services from a cloud provider and scale the resources on-demand. Actually, this underlying infrastructure, the datacenter hardware and software, is what is called a cloud [13]. Delivering a service at an abstraction level, which can be accessed by users on-demand by themselves, follows the Everything as a Service (XaaS) model [75]. The following three types are commonly known [63, 75]:

Software as a Service (SaaS)
delivers service in form of web applications [63]. Prominent examples are the Cloud Office Suites *Google Docs*[1] and *Microsoft Office 365*[2].

Platform as a Service (PaaS)
delivers service in form of a programming interface on which web applications can be developed [63]. Because details about how data is stored or requests are processed are hidden by the interface, such applications can easily be executed on different scales of computing power and storage increments. *Microsoft Azure*[3], for example, delivers PaaS [75] for several programming languages, such as *.NET*, *PHP*, or *node.js*.

[1]http://docs.google.com
[2]http://www.microsoft.com/ofice365
[3]http://www.windowsazure.com

Infrastructure as a Service (IaaS)

delivers service in form of an IT infrastructure and looks much like physical hardware [13]. For example, *Amazon EC2*[4] is a service providing VMs on which users can (nearly) control the entire software stack [13]. Furthermore, cloud storage belongs to this category [63] which delivers data storage as a service. Examples are *Amazon S3*[5] and *Dropbox*[6] that can be used by end-users to store and access data from "everywhere".

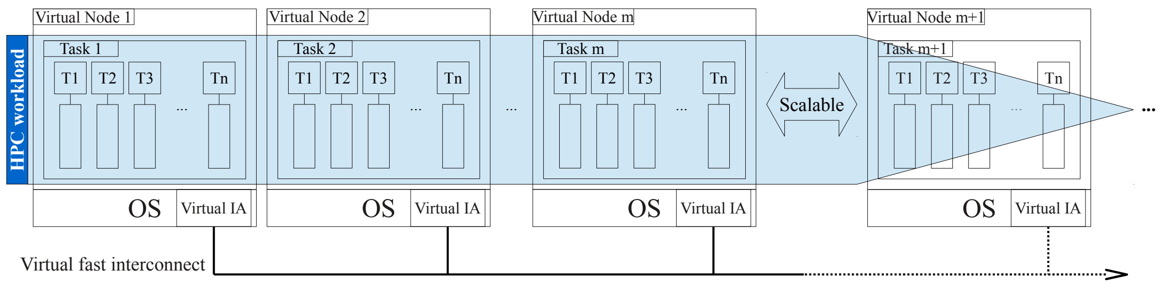

Figure 2.2: HPC IaaS pretends on-demand scalable HPC clusters to its users

HPC IaaS itself is understood as a special type of Infrastructure as a Service. This service is implemented by using machine virtualization as well as interconnect network virtualization (see Figure 2.2 and Figure 2.4 on page 9) [21]. It provides users an on-demand scalable virtual HPC cluster which can be typically scaled on amount of virtual nodes, as well as memory and storage per node.

2.1.1 Challenges of Virtualization for HPC IaaS

To deploy HPC IaaS, every HPC node and the interconnect of a virtual HPC cluster are virtualized. The main goal is to pretend an independent and scalable cluster that acts closely like a physical owned one. However, the requirements of HPC workloads differ significantly from typical server and workstation workloads which requires special optimizations in the virtualization setup. Typical HPC applications are compute intensive and task synchronization points often require equally placed compute units in the whole setup [21]. Also these workloads have higher demands on the underlying resources and on their guaranteed and timely delivery [21].

Virtualization is done by a hypervisor, also named as Virtual Machine Monitor (VMM) [44, 61]. It is a small program that virtualizes computing time, memory space, and I/O devices and therefore safely multiplexes the VMs onto the physical machine. Unfortunately, virtualization introduces overhead that potentially slows down the execution speed of a guest

[4] http://aws.amazon.com/ec2
[5] http://aws.amazon.com/s3
[6] http://www.dropbox.com

system compared to its speed running on a bare-metal system. Major keys to improve the guest performance are reducing the time of a processing unit spending in the VMM [15], minimizing the number of VMM traps needed by virtualization, as well as execute the VM, as long as possible, directly on the hardware [76]. This overhead is noticeable as delays in the VM execution, also called VMM-noise. Figure 2.3 provides an intuition on the potential effects of delays on workloads that are running with barrier-synchronized computation phases [60]: If noise introduces impact on a single thread, the computation phase of this thread is delayed and it will probably reach its synchronization point later than other threads of the workload. In such a case, the whole workload will run by this delay slower [60]. The likelihood of having at least one delayed thread per iteration increases, when the workload runs a large number of threads [60]. So, depending on the HPC workload, VMM-noise can cause fatal overall performance impacts and limits the scalability of parallel applications significantly [21].

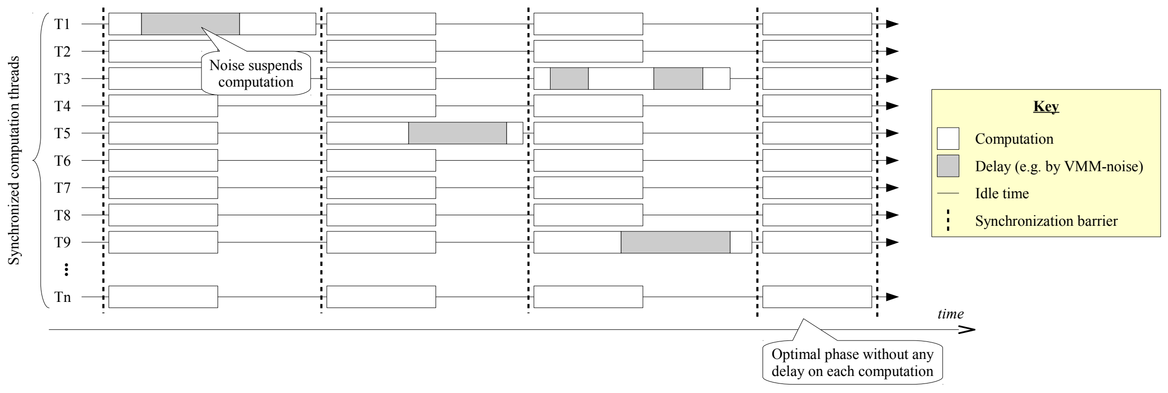
Synchronized computation threads
T1
T2
T3
T4
T5
T6
T7
T8
T9
Tn
Noise suspends computation
Key
Computation
Delay (e.g. by VMM-noise)
Idle time
Synchronization barrier
time
Optimal phase without any delay on each computation

Figure 2.3: Illustration of the impact of noise on synchronized computation (source: [60])

Most recent and popular processors, including the *x86* models from *Intel* [27, Volume 3, Chapter 23: Introduction to Virtual-Machine Extensions] and *AMD* [2, Chapter 15: Secure Virtual Machine], provide hardware virtualization features that can be utilized by the VMM to decrease some of these performance costs [15]. Most compute unit instructions can be executed natively in the guest, with the exception of a few privileged operations that are trapped by the VMM [40]. However, this overhead carries almost no weight because typical HPC workloads seldom call them [40].

Furthermore, for instance, Hillenbrand et al. [21] reduce jitter, caused by timesharing and memory overcommitment, by assigning dedicated compute resources to a virtual node for exclusive access. Also memory assigned to a virtual node is completely allocated from physical memory of the VMM host which avoids preemption and paging activity by the VMM. Kocolski et al. [33] even implement a dual stack virtualization approach to fuse HPC workloads with traditional workloads on a VMM and to retain workload consolidation [33]. For this purpose, a single system is partitioned into zones to provide isolated environments for the different workload requirements [33]. Each zone has its own underlying system software to provide optimal behavior for its zone [33]. For example, a NUMA-based multi-processor

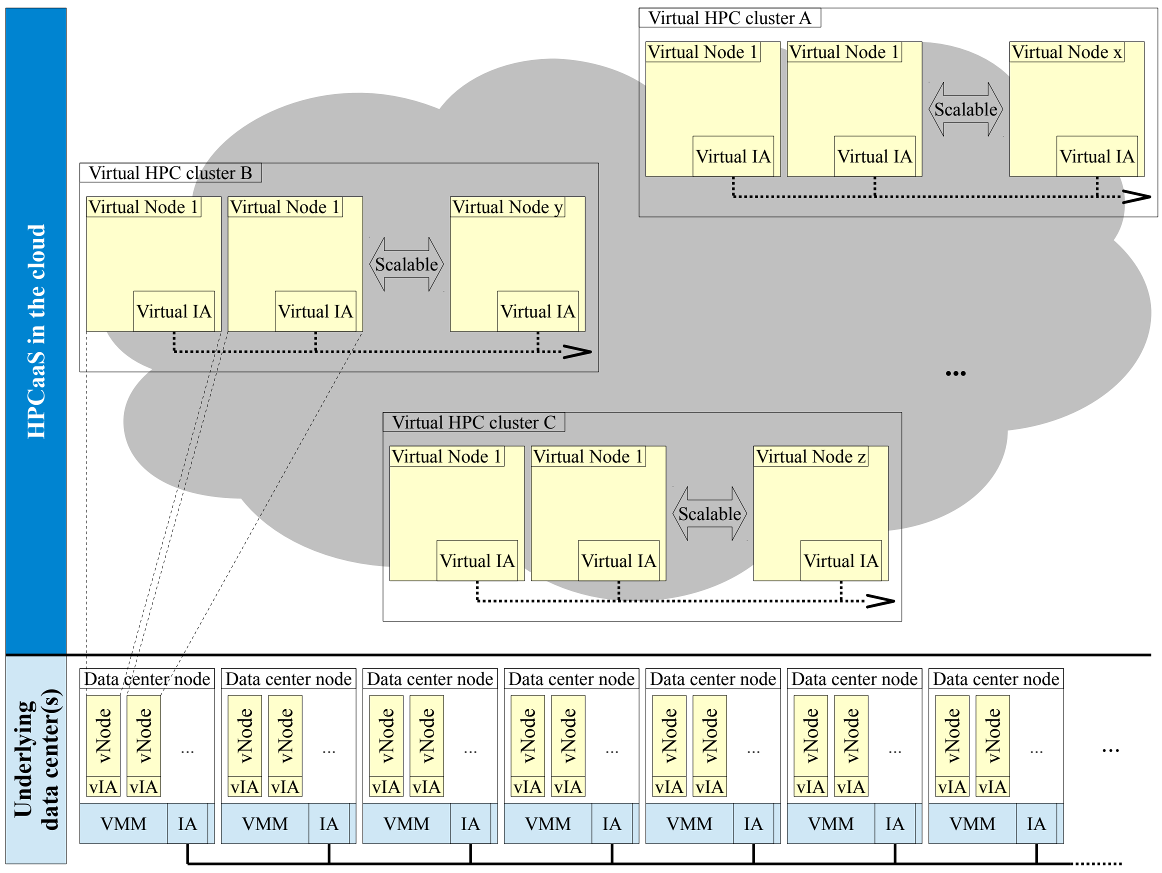

Figure 2.4: HPC IaaS is implemented by using machine virtualization

system is partitioned in such a way that an HPC VM is executed in an own HPC context on pre-selected processors and memory regions, while traditional applications are executed simultaneously on the rest of processors and memory [33]. The partitioning of the system is done in such a way that memory traffic is restricted as much as possible to the memory contained in the NUMA zones of the corresponding processors and their contexts [33]. Such an arrangement ensures that cross VM interference is minimized [33].

On the one hand exclusive resource usage leads to a decreased degree of virtual machine consolidation per VMM host, but on the other hand, due to the compute-intensive characteristics of HPC workloads, consolidation aspects are less relevant to HPC virtualization [44]. These aspects become beneficial whenever consolidation can be done with underutilized VMs running, for instance, traditional workloads.

2.1.2 VMM-based I/O Virtualization

I/O virtualization (IOV) is still a challenge for non-HPC virtualization, especially for network devices [53]. Moreover, it becomes a central role in HPC IaaS: Former research, such as Kocolski et al. [33] and Gupta et al. [19], identified I/O network performance as the main potential performance bottleneck. Gupta et al. [19] even evaluated several recent IaaS clouds by running communication-intensive HPC workloads on them. Their results show that cost benefits of Cloud Computing versus operating a physical cluster are overturning in this scenario, caused by poor network performance. Furthermore, supercomputers are typically built with special interconnect technologies [79], such as *InfiniBand* [25] and *Myrinet* [47] (also see Section 2.3 on page 20). These interconnects match HPC demands of minimum guaranteed bandwidth and bounded access latency [21], as well as low performance overhead. They provide cutting-edge communication performance by using aggressive protocol offloading, OS-bypass technologies (operating system bypass), and advanced features such as remote DMA (RDMA). For instance, *InfiniBand* as such hardware, show in comparison to typical *Ethernet* adapters lower latency and higher throughput [29]. Further, OS-bypass technologies offer hardware interfaces directly to HPC workloads that can be used without any OS-involvement during communication [21]. The OS is only involved in establishing connections, registering memory buffers, and to ensure protection [21].

Because of these reasons, knowledge about IOV technologies performance and overhead is vitally important for deploying HPC IaaS. Additionally, HPC workloads running on a virtual HPC cluster would optimally benefit from direct access to communication buffers of the physical hardware. In such a case, an HPC thread can circumvent all layers of system software during interconnect communications [21]. Generally, recent IOV techniques that are provided by the VMM can be classified into four different characteristics [39] which have different advantages and disadvantages measured against different disciplines. Further, Table 2.1 on page 12 shows a short comparison overview of these techniques.

Emulated devices

An emulated device, also called full virtualized device, simulates completely an existing hardware in software. Therefore, the VMM traps I/O operations issued by the guest to such a device and completely emulates its behavior [76]. I/O operations may also be issued on a connected physical, possibly different, device by the VMM [82] if configured. For instance, packets from and for emulated network devices will be sent and received through a physical network adapter in the VMM host by using the MAC address of the emulated device [76].

Emulation is intended for compatibility [76], thus for use with guests that come already with a driver software for this device and for those guest where para-virtualization (see next technique) is not applicable. However, in terms of efficiency this technique offers the worst performance in comparison [39]. The behavior of the physical hardware has to be completely emulated by the VMM. Guests do not get any access to a underlying hardware directly. But on the other hand, full device emulation is applicable for all types of devices [39] and because only a pure software instance is instantiated per device emulation instance, the number of emulated devices per VMM can easily be scaled [39].

Para-virtualized devices

On para-virtualizition, guests are aware of virtualization. Para-virtualized devices are intended to reduce dramatically the overhead caused by device behavior emulation (compare with previous technique). They employ an optimized pure software interface to the guest. Guests still do not have any direct access to an underlying hardware and I/O operations may also still be issued on an connected physical device by the VMM if configured. But various optimization of the VMM-guest interface in former researches [36, 70] promises huge performance improvements compared to traditional device emulation.

Unfortunately, recent implementations are focused on para-virtualized interfaces for block devices (e.g., storage), char devices, and network devices [39], but similar to device emulation, the number of para-virtualized devices per VMM can easily be scaled [39].

Accelerated para-virtualized devices

Accelerated para-virtualized devices are similar to para-virtualized devices, but provide guests direct access to I/O buffers of a physical underlying hardware [70]. It is commonly implemented with shared pages, which avoids copying of data between virtualized and physical device [53, 70].

For example, OS-bypass features of different I/O hardware can be utilized by the VMM to implement VMM-bypass [23, 40]. Another example is *Intel's* Virtual Machine Device Queues (VMDq) technology, which provides guests direct access to hardware network work queues [6, 53].

Performance measurements of systems implementing this type of technique [6, 40, 70] show that near direct I/O performance can be achieved [70]. Unfortunately, availability of this technique depends on the I/O hardware and the VMM which has to support the feature of this hardware [39]. Also the number of devices per VMM is limited by the hardware [39].

Device pass-through

Device pass-through, also called direct device assignment, means that a VM sees and interacts with a real device of the VMM host without software intermediary [82]. Main advantages are that nearly bare-metal performance is possible [15] and all device features are accessible by the guest.

However, hardware support in form of an IOMMU, an MMU for I/O devices, is at least required to ensure overall system security and stability [81]. It can pretend a device from accessing memory regions which are not belonging to its assigned guest, thus retaining the isolation between the VMs. In virtualized environments VMs have their own view of physical memory which typically distinct from the VMM host physical memory [82]. The IOMMU, such as *Intel's VT-d* [26] or *AMD's IOMMU* [1], can be

Technique		*Characteristics*	
	Efficiency	**Applicability**	**Scalability**
Emulation	Low	All devices	High
Para-virtualization	Medium	Block, network	High
Accel. para-virt.	High	VMM dependent	Medium
Device pass-through	High	All devices	Low

Table 2.1: Comparison overview of IOV techniques (source: [39])

programmed by the VMM to translate physical memory addresses into guest physical addresses for each assigned device.

A limitation of bypassing the virtualization layer [82] is that the VMM looses completely control of the device and cannot observe the device state which imposes challenges to VM migration and device monitoring (we will go in a more detail in Section 2.2 on page 15) [53, 70, 83]. Also recent implementations require the pinning of the entire guest memory as long as no para-virtualized interface is implemented for the guest to map and unmap guest physical memory regions for a device [15, 72]. On the other hand, entire pinning introduces only minimal overhead [15] and we pointed out that, in terms of HPC IaaS, an approach to avoid paging activity of guest physical memory by the VMM pins this memory as well. Unfortunately, an assigned device cannot be shared with other VMs because the intermediate layer, which could perform resource multiplexing, is removed.

In terms of HPC IaaS, accelerated device para-virtualization and the device pass-through technique will perform well for virtualizing interconnect hardware since they permit HPC VMs to directly access memory buffers of the underlying hardware. These techniques further introduce only a minimal virtualization overhead and fulfill the major keys to improve HPC IaaS performance: reducing the time of a processing unit spending in the VMM and minimizing the number of VMM traps needed by virtualization (review Section 2.1.1 on page 7). Especially, device pass-through allows VMs to directly access I/O devices, without VMM intervention for data movement resulting in high performance [35].

2.1.3 Self-Virtualizing Devices

Recent hardware enhancements move more and more I/O virtualization logic from the VMM to the hardware [68], mainly to increase performance by reducing software virtualization overhead [28]. Early studies on self-virtualizing devices [68, 80], which appear as multiple separate interfaces, promised great performance improvements compared to software-based IOV techniques. Such natively shareable devices also defeat the drawbacks of direct device assignment, such as device sharing, potential underutilization, and scalability [35, 39]. These devices typically provide unique memory space, registers, and interrupts for each exposed virtualized device while utilizing shared resources in a resource sharing logic [28].

The former approach of Raj et al. [68], as well as the industrial standardization Single Root I/O Virtualization (SR-IOV) [59] by PCI-SIG [58], an organization that releases and develops the PCI [57] and PCI Express [55] standard, introduces an unequal device exposing scheme. This scheme distinguishes master devices, also called physical functions (PFs), from slave devices, also called virtual devices or virtual functions (VFs). Primarily, the master device provides a special interface to manage and configure multiple slave devices and a slave device is typically controlled by exactly one master device. Furthermore, each slave device is a lightweight device that is intended to be assigned to a VM. It owns resources necessary for data movement, including DMA, and a carefully minimized set of configuration resources [28]. In terms of device drivers, this distinction leads to separate drivers, one for master devices and one for slave devices. They do not have to be necessarily split into multiple driver programs, we also observed combined drivers that execute separate program flows depending on the device type (compare with *Mellanox ConnectX-3* driver architecture in Section 4.2.2 on page 48).

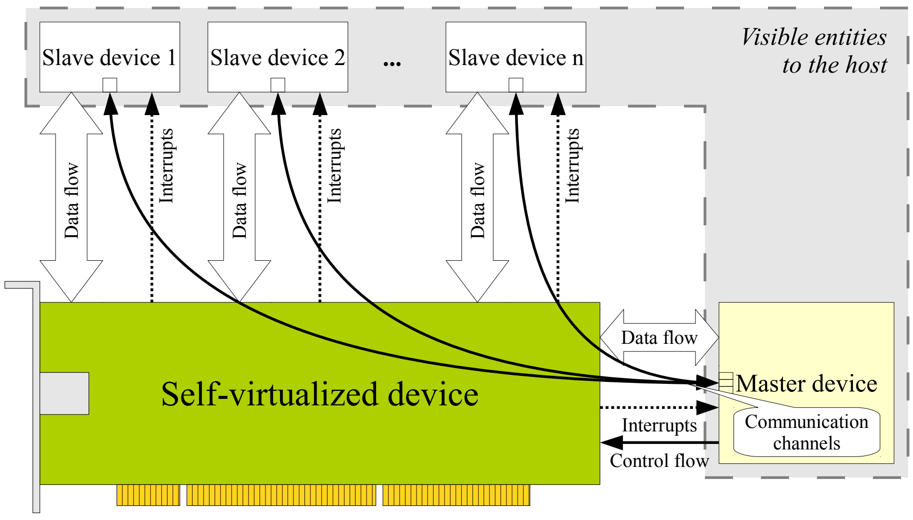

Figure 2.5: Principle of a self-virtualizing device using unequal device exposure and communication channels

In the SR-IOV standard, the basis on that we later introduce our migration approach, a slave function needs only hardware for accessing performance critical resources while non-performance critical resources are emulated by the master driver [10]. The idea is that the master driver keeps control over the hardware while slave devices are lightweight access interfaces to performance critical resources. For this purpose the SR-IOV standard suggests to implement a communication channel between slave driver and master driver [28, 59]. VMM independence is achieved by a hardware-based or hardware-assisted implementation. With this channel, a slave driver is able to request operations through the master driver, as well as able to receive event notification from the master [10]. The master can then inspect slave requests and enforce policies concerning performance and security isolation as well as manage and monitor the resources used by slaves [10]. For example, a slave may request operation that would have global effects [28] or would affect isolation between VMs. In such a case the master driver could take appropriate actions or even simulate a behavior. Furthermore, the master driver may also need to notify changed resource status to its slaves (e.g., network port link status change) [10].

The actual resource sharing logic and management is implemented in hardware or software or even both. Typically, only the master driver has access to it for configuration (e.g., to assign an I/O buffer to a slave device). The implementation of the logic depends highly on the resource technology itself and on design decisions by the vendor of a specific hardware. For example, self-virtualizing *Ethernet* devices typically implement a physical Layer 2 switch with sort logic, to place incoming packets on receive queues dedicated to its target slave device [28, 68]. We will provide a more detail of *InfiniBand* interconnects in Section 2.3 on page 20.

In conclusion, using device pass-through with self-virtualizing devices has the following benefits: (1) (almost) no VMM interceptions needed when VMs are performing I/O on slave devices, (2) virtual guest can use device-specific hardware features that are only limited by the vendors design, and (3) VMM still would have the ability to control device sharing via a master driver interface. Unfortunately, assigned self-virtualizing devices still impose challenges to live migration, which is one of the most important virtualization features [83]. We go in a more detail in Section 2.2 on the next page.

2.1.4 Interconnect Network Virtualization for HPC IaaS

Besides virtualization of the node resources computing time, memory, and I/O, each virtual HPC cluster should get the impression of using the interconnect network by itself [21]. Therefore, each virtual interconnect has to be isolated from others running on the same cloud. In general, the basic concept is similar between HPC IaaS and IaaS besides higher requirements on HPC IaaS, such as minimum guaranteed bandwidth and bounded access latency [21].

Interconnect network virtualization, or network virtualization in general, is typically done by partitioning the underlying interconnect network to isolate network traffic. Every virtual interconnect adapter is associated to a single partition and all nodes of a virtual cluster are bound to the same partition. However, how partitioning is implemented in detail, depends highly on the deployed interconnect technology and its available isolation techniques.

2.2 Challenges of Live Migration for HPC IaaS

VM migration is seen as one of the most powerful features of virtualization [83]. It is used to move VMs to different physical nodes, for example for maintenance, high availability, consolidation, or load balancing. For these reasons, VM migration is an important tool for running and maintaining a cloud data center for IaaS (thus, for HPC IaaS, too). There are various different definitions of migration types [17, 64]. We categorize VM migration into cold and live migration (also called hot migration): Cold migration stands for the traditional way to migrate a VM by shutdown, moving its storage (if necessary), and then restarting the VM at the destination host [17, 64]. Live migration means relocation of a running VM to a destination host. Thus, it involves stopping the execution of the original VM, moving the VM runtime state to the migration target, adapting the configuration of the interacting environment, and at last resuming VM execution at the destination. Furthermore, a guest-transparent migration must not involve a guest into state restoration during migration. It has to be done completely at VMM layer.

We examine the feasibility of extracting and reconstructing a slave device state of a self-virtualizing hardware at VMM layer in this thesis. Therefore, we give an overview of migration itself in this section. First, we discuss in Section 2.2.1 which components and states are included by a VM live migration. After that, we provide more detail about several migration procedures in Section 2.2.2 on page 17. Several designs of hardware that is migrated can add further requirements and dependencies to migration. For instance, such a design involves remote hosts in the environment configuration adaptation process. Therefore, we introduce two different kinds of migration strategies: (1) common migration approaches, which are applicable whenever hardware migration can be implemented transparent to interacting parties, and (2) cluster checkpoint approaches, whenever this transparency cannot be achieved.

2.2.1 State involved in Live Migration

A VM migration moves the machine state to a different host and adapts the configuration of the interacting environment, so that all involved parties can continue operation (almost) seamlessly after migration.

Adapting the configuration has to be done in order that all parties that are interacting with the VM retain connectivity and reachability to the VM after a migration. In terms of networking, this means that services provided by the VM are reachable after a successful migration in the same way as before and communication connections are maintained during the migration process or reestablished. For instance, approaches using conventional *Ethernet* migrate the MAC address of virtual network interfaces together with the VM instance. After the migration proceeding, communications stay alive as long as the migration destination is connected to the same subnet as the original machine [48]. In contrast to that, some popular interconnects in HPC, such as *InfiniBand*, are managed by a dedicated network instance that sets up routings and end point node addresses [25]. These addresses are bound to the physical hardware, and thus, they cannot be migrated with the VM. In such a case, all involved parties in connectivity have to adapt to the new situation.

Like Nelson et al. [48], we differentiate between three kinds of VM state:

Virtual device state
State of virtual devices of the VM, such as computing unit, system devices, graphic adapters, network adapters, etc.

Location dependent resources
Location dependent resources are resources on the VMM host to which virtual resources are connected. For example, virtual networking is often associated with real network hardware that enables VMs to communicate with hosts on other physical machines.

Guest memory
Memory that is seen as physical memory by the guest.

In general, guest memory and virtual device states are just copied to the destination. Connections of virtual resources are reestablished to the new host-local resources. This reestablishment of connection is dependent on the resource itself. Earlier works propose solutions to solve issues on common full-virtualized or para-virtualized I/O, such as virtualized *Ethernet* networking and storage [7, 48]. However, despite former research on live migrating VMs configured with a pass-through device [17, 23, 30, 53, 71, 83], there is no generic approach available yet. As described in Section 2.1.2 on page 10, the VMM loses control of an assigned device and is typically not able to copy the device state to the target. Internal device states may not be accessible and state transitions may be still in flight at migration time [83]. But, as we pointed out, better results are achieved in terms of overall performance and lower virtualization overhead, whenever VM gets direct physical access to the device. However, the more state data is located or visible at the VMM layer, which applies for example to emulated and para-virtualized devices, the straighter forward should be the implementation of migration support. These correlations seem to be in conflict. Zhai et al. [83] identifies that approaches for device migration have to be addressed either with a VMM having device knowledge or with guests that are involved into migration to perform the device reinitialization. In the second case, a guest becomes aware of migration and thus, such approaches are not guest-transparent [17, 23, 30, 53, 71, 83]. Giving up guest-transparency restricts users to run modified driver, kernels, or even libraries in the guest that are implementing the mechanisms needed for reinitialization. Only they can guarantee errorless operation to the users, whenever migrations are performed in the data center.

However, as we show in Chapter 3 on page 27, self-virtualizing devices implementing an unequal device exposure (see Section 2.1.2 on page 10) can turn back control to the VMM while retaining the performance benefits of device pass-through. We consider these self-virtualizing devices as a hardware-implementation of accelerated para-virtualization, where a VMM can manage and control the resources of slave devices by using the master device interface.

2.2.2 Live Migration Strategies

Like Clark et al. [7], we consider live migration as a transactional interaction between two VMMs (in the following: Host A and host B) that are involved into the migrating a VM. This interaction is done by a process of five common stages starting on host A that runs the VM that will be migrated to host B (also shown in Figure 2.6 on the next page):

Stage 1: Pre-Migration
A target host (called host B in the following) is selected that guarantees resource availability of host-local resources that are equal to the host-local resources currently connected to the VM [7].

Stage 2: Reservation
Host B confirms that the requested host-local resources are available and creates a VM container for the migration [7].

Stage 3: Stop-And-Copy
Host A stops the VM execution and copies the VM state into the container on host B [7]. Host B also establishes connections to its host-local resources. Actions to adapt the configuration of the interacting environment by the VMMs may also take place in this stage. At the end of this stage the VM is cloned to host B and both instances is stopped.

Stage 4: Commitment
Host B indicates to host A that the image was successfully copied and its copy is in a consistent state [7]. After that, host A may now discard its VM instance that was hold to resume it in case of migration failures [7].

Stage 5: Activation
Host B resumes VM execution.

Various improvements to the above-mentioned *stop-and-copy* approach were introduced to reduce the long service downtime due to the copy process of memory while the VM is stopped. Prominent examples are *pre-copy* [7], *pure demand-migration*, *post-copy* [22], and even combined hybrid techniques (see [7, 22] for an overview). These improvements try to shift a huge part of the memory page copy process into stages where the VM is running. Only a required minimum is then transferred at the Stop-And-Copy stage. For example, *pre-copy* copies memory pages in several iterations starting with copying all memory pages and followed by multiple iterations copying modified pages while the VM remains running prior the actual Stop-And-Copy stage. In doing so, the typical downtime is reduced, since only a small set of remaining modified memory pages has to be copied when the VM execution was stopped. *Pure demand-migration* migrates only a minimal required set at the Stop-And-Copy stage and transfers remaining memory pages on-demand on their first access after the VM is resumed on host B.

Nevertheless, recent improvements are not feasible with direct device assignment on common hardware platforms. For example, *pure demand-migration* suffers on devices that need

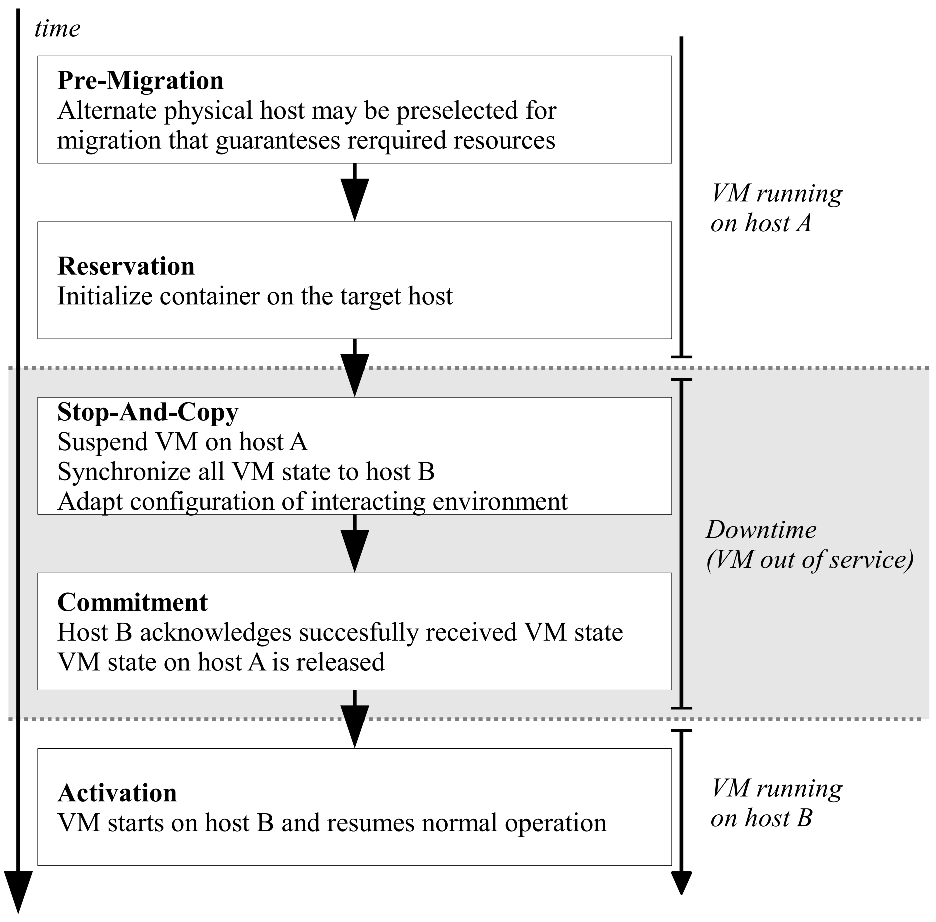
time
Pre-Migration
Alternate physical host may be preselected for migration that guaranteses rerquired resources
Reservation
Initialize container on the target host
Stop-And-Copy
Suspend VM on host A
Synchronize all VM state to host B
Adapt configuration of interacting environment
Commitment
Host B acknowledges succesfully received VM state
VM state on host A is released
Activation
VM starts on host B and resumes normal operation
VM running on host A
Downtime (VM out of service)
VM running on host B

Figure 2.6: Five common stages of live migration (derived from [7])

to access data from guest memory pages that are not already migrated. Dong et al. [11] and Pan et al. [53] identified missing dirty page tracking in existing IOMMU that is needed to track memory pages dirtied by device access, such as DMA, and which is needed by *pre-copy migration*. Migrating an obsolete page state can cause fatal guest malfunction or even system crashes. To avoid this, the VMM needs knowledge of potential memory pages accessed by the device via device knowledge and resource monitoring or it has to view the whole guest memory as dirty. This in turn makes memory migration optimizations useless. Another problem may occur whenever the assigned device does not provide any suspend mode that still allows access by a driver to load and store the hardware state. Especially on devices using RDMA, it is difficult to get the device in a quiescent state to perform a hardware state migration. In these cases, the hardware may still write to guest memory even when the VM execution is stopped. But we believe that in most cases, when using pass-through of self-virtualizing devices using unequal device exposure, it is possible to utilize some device specific operations in the VMM to bring a slave device into a quiescent state. In this state, the slave device stops accessing guest memory which ensures that the VMM is able to migrate guest memory pages in a consistent state.

In addition, the presented approaches rely on the assumptions that interactions with the VM are using an interconnect technology that can handle temporary service downtimes and that adapting the environment configuration is transparent to interacting remote parties. The second assumption is not necessarily valid in HPC: For instance, in *InfiniBand* interconnects, end point node addresses are managed by a dedicated network instance and bound to the physical hardware [25]. As long as such addresses cannot be migrated with the VM in any way, interacting remote parties have to be involved in the reconfiguration process of the interconnect (e.g., by reconnecting to a new end point address). Some alternatives to pure VM-local considerations utilize checkpoint procedures, such as *Checkpoint-restart*. *Checkpoint-restart* of a virtual HPC cluster, as proposed by Scarpazza et al. [71], involves suspending the whole cluster and bringing it to a save point before further changes, such as VM migration, are made. On failure, the environment can continue operating from its last checkpoint. The migration itself is performed within a global silence phase where the HPC cluster is completely stopped and all outstanding communication operations are completed (including RDMA operations). This approach eliminates many side effects and reduces the implementation complexity of the migration procedure of a VM. For instance, quiescence of cluster interconnect devices, as we pointed out as requirement for special hardware in Section 2.2.1 on page 15, is achieved automatically. No stopped VM is able to initiate new communication operations (including RDMA). Furthermore, service downtime due to migration plays a minor role, since stopped VMs do not access services (e.g., tasks of HPC workload) of migrated VMs any further. The actual environment adaption can then be performed by all involved VMMs. Scarpazza et al. [71] describe *Checkpoint-restart* procedures with a three-phase model:

Phase 1: Drain
The Drain phase begins with the suspend notification to the virtual cluster and covers the time the cluster needs to reach global silence. It includes successfully suspending all VMs as well as completing outstanding communication.

Phase 2: Global Silence
At this phase it is possible to simultaneously migrate multiple VMs to different VMMs and to perform interconnect reconfiguration.

Phase 3: Resume
The Resume phase covers the time the cluster needs to go back into normal operation. All VMs are resumed in this phase.

However, this approach does make downtime only invisible to interacting parties when they are all under the control of a coordinating instance. This instance has therefore to be able to suspend each party. Generally, these assumptions apply for HPC IaaS, where each HPC node is virtualized and the according virtual cluster is managed by a data center management instance. Furthermore, compared to traditional cloud jobs, HPC jobs are typically isolated from third party communication and communication is used only within the cluster for synchronization and intermediate stage coordination purposes.

2.3 InfiniBand as HPC Interconnect

InfiniBand is currently a popular and modern interconnect for HPC which is also reflected by the TOP500 supercomputer list of November 2012 [79]: *InfiniBand* is used in 44.8% of all listed installations and the most used one, even amongst the 15 highest ranged supercomputers. It is mainly designed for building system area networks (SANs). Such a SAN is intended, like shared bus architectures, to connect processors nodes and I/O nodes together [25]. However, *InfiniBand* resolves scalability, expandability, and fault tolerance limitations of bus architectures by utilizing bidirectional point-to-point links through switches and routers [9]. Because *InfiniBand* utilizes massive protocol offloading, zero processor-copy data transfer (also called RDMA), and OS-bypass techniques [25], it provides cutting edge communication performance and low latency that meet the requirements for communication-intensive HPC workloads running on multiple processor nodes [21].

In the following sections we give a brief overview of the *InfiniBand* architecture specification [25] which was published by the *InfiniBand Trade Association* in November 2007. After that we discuss in Section 2.3.5 on page 25 how self-virtualization is implemented on common *InfiniBand* host adapters.

2.3.1 Subnets and Network Management

InfiniBand networks are organized in subnets, also called fabrics [73]. As represented in Figure 2.7 on the facing page, they consists of multiple *InfiniBand* endnodes that are interconnected with bidirectional point-to-point links through switches or routers or are interconnected directly [16]. Such an endnode is defined as a device, other than a network switch or router, and is connected to the fabric with a channel adapter (CA) [73]. Such a CA may have multiple ports. A CA of a host device is called host channel adapter (HCA) and a CA for an I/O device, like a storage subsystem, is called target channel adapter (TCA). Each CA, and also its ports, is uniquely identifiable with persistent global unique IDs (GUIDs). These GUIDs are assigned by the CA vendor during manufacturing [25]. Switches route traffic within its local subnet and router route traffic between different subnets [73].

For each *InfiniBand* subnet one subnet manager (SM) entity is responsible for configuring and managing switches, routers, and CAs. When multiple SMs are active in a subnet, the SM instances negotiate which of them will operate as master SM [16]. Such a SM is typically implemented within a fabric device, more precisely on a CA or a switch. For instance, *OpenSM* [51] is a popular open source software SM that can be run on a host configured with a HCA. The master SM checks periodically, or on a incoming signaling message (trap), the fabric for changes and assigns each port of every CA connected to the subnet with a local ID (LID) and the subnet prefix. The LID is unique within the subnet and the subnet prefix addresses the according subnet. Further, the master SM performs route calculations and configures the switches of the according fabric [16]. Routers connected to the subnet are informed about the subnet by the SM. Each CA, router, and switch of the fabric run

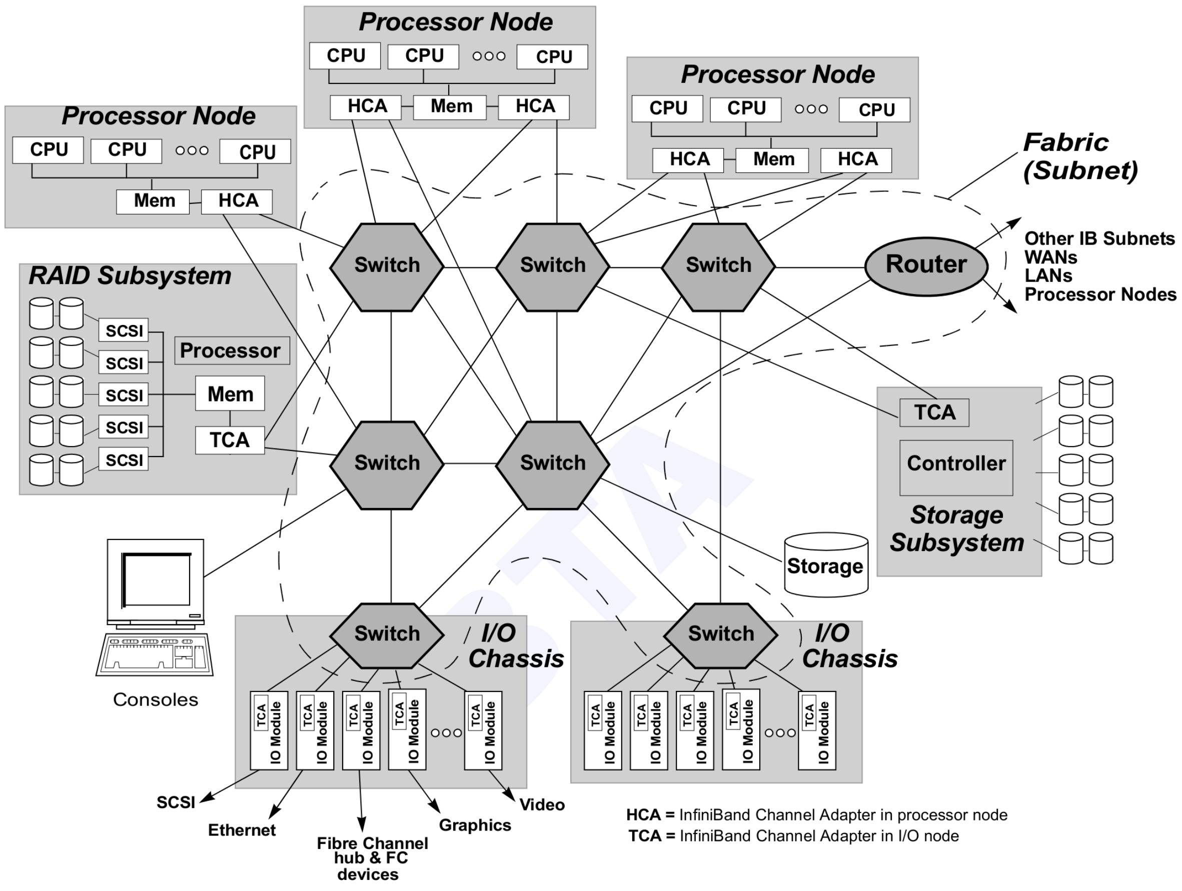

Figure 2.7: Exemplary *InfiniBand* network overview (source: [25])

a low level functionality which is called subnet manager agent SMA that responses to the commands and queries sent by the master SM [9].

Within a subnet, remote CA ports are addressed with their LID. Outer-subnet addressing, however, works a bit different: Routers are not completely transparent to the endnodes since the source CA needs to specify the LID of the router port in the subnet and also the global ID (GID) of the destination. This GID is assembled with the target port GUID and the assigned subnet prefix.

2.3.2 Queue Pairs and Transport Services

Communication over *InfiniBand* interconnects takes place between two queue pairs (QP). They form a virtual interface of the CA hardware to the consumers (e.g., an application on a processor node). Each QP is isolated and protected from other QPs and can be considered as a private resource assigned to a single consumer. Such a QP consists of two work queues (WQ) where a consumer can post work queue elements (WQEs) on it: (1) a receive WQ

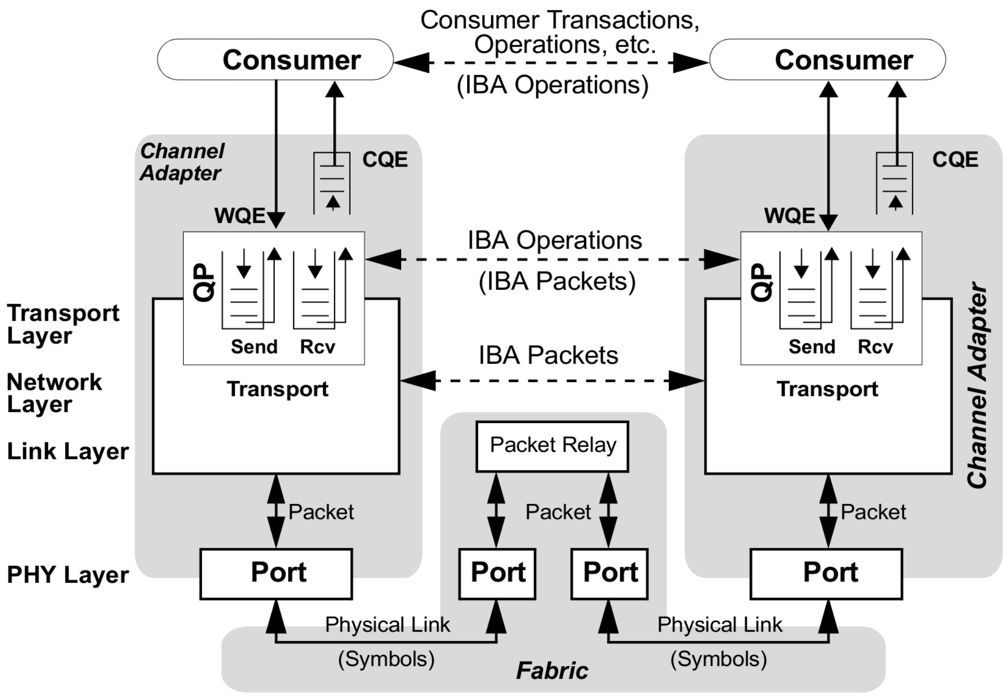

Figure 2.8: Communication stack of *InfiniBand* (source: [25])

and (2) a send WQ. Such a WQE describes a work request and is picked-up (according to the FIFO principle) for execution by the CA [9]. The request is either a sending operation request on the send WQ (e.g., data send, RDMA access) or a receiving operation request on the receive WQ. However, there is only one receive operation currently available and it is to specify a receive data buffer for data that will be received by a remote consumer executing a send operation. When a CA completes a work request, it places a completion queue element (CQE) on a associated completion queue (CQ) of the QP. This way, the status result of a WQE is reported by the hardware. Such a CQ can be associated to multiple work queues and has to be created by the consumer beforehand. The Figure 2.9 on the next page provides an illustration of this QP interface concept.

Each QP instance is associated with a so called queue pair number (QPN) which uniquely identifies it on a CA. There are only two predefined QPs that exist for each particular port of a CA: QP0 is dedicated for the SMA instance of the CA as well as for a possible SM application instance running on the according node. QP1 is reserved to be used by special management service agents (on further interest, we refer to the *InfiniBand* specification [25]).

Before a consumer can use a QP and start to communicate through the *InfiniBand* fabric, it must first create a QP instance and specify its class of transport service. In connection oriented communication, each QP is bound to exactly one other QP usually on a remote node. Further, a QP rejects every WQE that is not valid for the configured service class.

Like *Ethernet*, *InfiniBand* uses packet based data transport on the interconnect but imple-

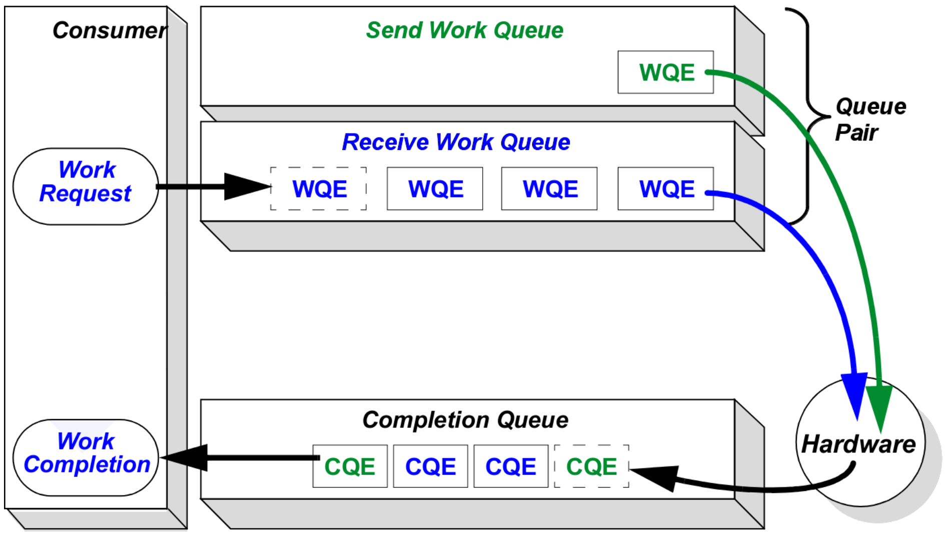

Figure 2.9: Queue pair (QP) driven work requests (derived from [25])

ments mechanisms for transport services classes within the CA. A consumer can thereby specify between, as shown in Table 2.2 on the following page, five different QP transport services during QP creation:

Reliable connection
: Reliable transfer of data between exactly two connected QP instances in the *InfiniBand* network [9]. They are comparable with TCP connection in the TCP/IP standard [54] which means that they include error recovery, flow control, and reliability due delivery acknowledgement.

Unreliable connection
: Unreliable transfer of data between exactly two connected QP instances in the *InfiniBand* network. However, messages may be lost or delivered out-of-order or even corrupted.

Reliable datagram
: A QP configured in this mode can just send and receive messages from one or more QPs without establishing a connection beforehand. Delivery of messages is error recovered and acknowledged.

Unreliable datagram
: A QP configured in this mode can just send and receive messages from one or more QPs without establishing a connection beforehand. Like unreliable connection, a datagram may also be lost or delivered corrupted. This type of transport service is comparable with UDP datagrams in the TCP/IP standard.

Raw datagram
: Raw datagrams are messages that are not interpreted and handled by the CA hardware.

Service type	**Connection oriented**	**Acknowledged**	**Transport**
Reliable connection	Yes	Yes	*InfiniBand*
Unreliable connection	Yes	No	*InfiniBand*
Reliable datagram	No	Yes	*InfiniBand*
Unreliable datagram	No	No	*InfiniBand*
Raw datagram	No	No	Raw

Table 2.2: Service types of *InfiniBand* QPs (source: [25])

Raw datagram QPs can be seen as an interface from the *InfiniBand* data link layer that is directly exported to the consumer [9]. The properties of raw datagrams are the same as unreliable datagrams: They can be sent and received from one or more QPs without establishing a connection beforehand. Such a datagram may also be lost or delivered corrupted.

The main purpose of raw datagram QPs are to send and receive messages for a consumer-handled protocol other than *InfiniBand*, for instance, IP or *Ethernet* packets [73]. This way, non-*InfiniBand* packets can be sent through a *InfiniBand* interconnect [73].

2.3.3 Remote DMA (RDMA)

InfiniBand supports RDMA operations that perform directly a write or read access to memory of a remote endnode. Such a memory access operation is directly executed by the according remote CA without further involvement or notification to the remote consumer. The consumer requesting a RDMA access thereto submits a RDMA WQE on a send WQ of a reliable connection QP or reliable datagram QP. Access permission is coordinated by the remote consumer with a prior setup of host memory regions that shall be accessible via RDMA. These regions are protected with pairs of memory keys, a local key (LKey) and a remote key (RKey), generated by the CA during memory region registration. The LKey is used internally in work requests to describe a memory region to a local QP. The RKey is passed to every consumer allowed to perform RDMA, which then have to supply this key in each RDMA WQE.

2.3.4 Fabric Partitioning

InfiniBand supports a way of subnet partitioning via partition keys (PKey). They are setup by the SM which assigns corresponding keys to each port of a CA. In the result, QPs, except QP0, QP1, and raw datagram QPs, are required to be configured for the same partition to be able to communicate with each other. PKeys are therefore carried in every packet of the

InfiniBand transport (see Table 2.2 on the preceding page) and compared with the configured PKey of the receiving entity. Received packets whose comparison failed are rejected.

2.3.5 Resource Sharing Logic of Self-Virtualizing HCAs

HCAs that implement self-virtualization using unequal device exposure (as introduced in Section 2.1.3 on page 12) typically assemble this with the (1) virtual switch or (2) shared port model. In the following we discuss these models based on a presentation by Mellanox's architect Liran Liss held on the *OpenFabrics Alliance* workshop in 2010 [39] and published patches by Mellanox [46] to the Linux RDMA development mailing list [38]. These patches add support for *InfiniBand* HCAs that implement self-virtualization with the shared port model.

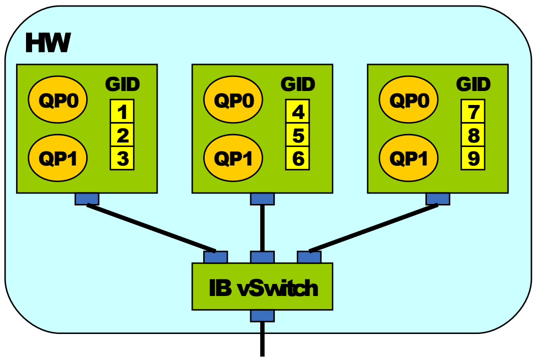

Figure 2.10: Slave HCAs (VFs) behind virtual switch (source: [39])

The *InfiniBand* virtual switch model (see figure 2.10) is comparable with virtual switch implementations of self-virtualizing *Ethernet* adapters. In this model, the *InfiniBand* HCA presents each slave device as a fully stand-alone HCA with own ports and own GUIDs to the interconnect. The ports are connected together on a virtual switch logic implemented in the physical HCA.

In the shared port model (see figure 2.11), however, the ports of the physical HCA get shared between multiple slave devices. In the point of view of the *InfiniBand* network, there are still only the physical HCA ports seen. Although, each slave HCA and slave port gets its own GUID generated by the master driver and assigned LIDs and subnet prefixes of the master's HCA ports are passed to the depending slaves. Thus, they are shared between master HCA and slave HCAs. Certainly, each virtual port GUID is registered as an alias at the master's physical port. So, each slave still has its own GID. Some other resources, however, are necessarily shared between the master and slave devices. For instance, the QPN space is shared between every HCA instance on a physical HCA and PKey configurations are shared between master HCA and slave HCA when using the same physical port. Since QP0 is coupled with the physical management of the subnet, QP0 is not exposed to slave HCA ports and is totally owned by the master HCA. Slave HCA ports still have a QP0 but no data is

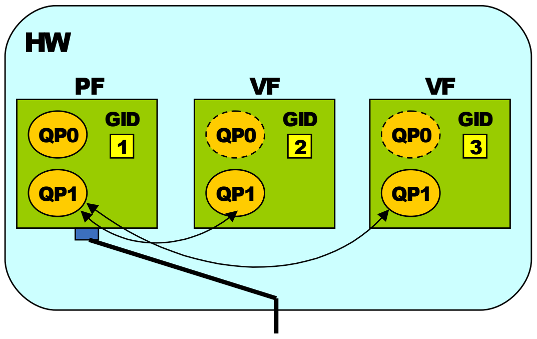

Figure 2.11: Shared port model (source: [39])

coming from and all data that is put into is be dropped. This implies that a SM cannot be run on a slave. All traffic to and from a slave's QP1 is tunneled through the master HCA where it can virtualize or forward certain requests. This way, the master HCA remains full control of *InfiniBand* management and eventually exposes QPs instances to slave devices.

As discussed by Mellanox, the shared port model seems to be more attractive for *InfiniBand* implementation, because it provides higher scalability and lower potential performance degradation compared to the virtual switch model: Since the *InfiniBand* network is unaware of slave HCAs, no additional LIDs are assigned per VM which would unnecessarily bloat the LID space when a huge bunch of VMs configured with slave devices present in a fabric. This way, no routing entries in switches are wasted and potentially associated caching effects in the *InfiniBand* network components that slow down packet forwarding are circumvented.

3 Design

In this chapter, we propose our approach and tool sets to enable guest-transparent live migrating virtual machines (VMs) configured with a directly assigned slave device of a self-virtualizing hardware. Despite we target our approach for High Performance Computing Infrastructure as a Service (HPC IaaS), we introduce our approach as generic as possible since some of the introduced concepts are also valid in migrating other device states.

This chapter is structured as followed: In Section 3.1, we begin with a brief and global overview of our migration approach and introduce our solution of live migrating in HPC IaaS. After that, we describe the actual and two orthogonal components of our approach for guest-transparent live migrating device states. The first component consists of migration strategies introduced in Section 3.2 on page 31. They are coordinating the migration process in a way to guarantee migration transparency to the guest and also to interacting remote nodes. Because these strategies are based on virtual machine monitors (VMMs) that are able to extract and restore the state of a directly assigned device, we introduce the second component in Section 3.3 on page 35, a novel interface at the master driver that we called migration assistance interface (MAI). This interface provides the necessary mechanisms to enable guest-transparent state migration of a slave device and provide them to the VMM. It thereto utilizes the self-virtualization to break up slave device state opaqueness. After that, we define specifically which state data is migrated for directly assigned slave devices in Section 3.4 on page 38. We also give some generic comments for implementing the migration. Finally, in Section 3.5 on page 40, we introduce a *InfiniBand* specific approach of slave device migration that is executed when the state is extracted and restored in our HPC IaaS scenario.

3.1 Overview

As we analyzed in the previous chapter, we observe two aspects of live migrating a pass-through device: migrating the hardware state itself (see Section 2.2.1 on page 15) and the migration procedure (see Section 2.2.2 on page 17).

We propose two different basic strategies, named local and global migration strategy, to perform a migration. The choice of a migration strategy (as well as the details of state migration) depends highly on the actual implementation of the migrated hardware. The local migration strategy performs the VM migration in the traditional way without any further involvement of any remote parties in the migration process. It can be used whenever the hardware state migration can be performed in a way transparent to interacting remote parties and the adaption of the environment can be induced by both VMM hosts that are involved in the migration. A global migration strategy involves suspending of all interacting remote parties of the VM that is migrated. These remote parties are also virtualized so that all VMMs can work cooperatively together to reconfigure the environment interconnect accordingly. Since all VMs are suspended, the reconfiguration process cannot affect any running VMs.

In both strategies, we perform guest-transparent hardware state migration by utilizing self-virtualizing devices implementing unequal device exposure, such as introduced in Section 2.1.3 on page 12. Such a device exposes a master and multiple slave devices where the master device is used by a special master driver to manage and configure the slave devices. A slave device is intended to be assigned to a VM and owns physical resources necessary for direct data movement which leads to bypassing the virtualization layers for direct access to the physical shared resource (e.g., a network). The master driver is typically running in a special privileged VM running drivers (e.g., Dom0 on Xen) or even within the VMM itself (e.g., the host Linux in KVM). For this reason, the driver is accessible by the VMM to assist migrating a pass-through slave device state.

We propose a novel and generic interface in the master driver that we call migration assistance interface (MAI). It is used by the VMM to perform state extraction and restoration of a pass-through slave device. However, the actual extraction and restoration is done by routines in the master driver. With this model, we break up device state opaqueness of assigned slave devices and follow the basic approach to address device migration with a VMM having device knowledge which was identified by Zhai et al. [83]. However, we give the VMM device knowledge in an indirect way. Device knowledge still remains in the device drivers, which makes our solution more applicable to a wider range of devices (as well as for proprietary and closed source drivers since implementation details are not carried out from the master driver). Figure 3.1 on the next page illustrates the utilization of MAI during a live migration: While the source VMM iterates through the virtual devices of the guest and transfers their state to the destination host, the VMM utilizes MAI to extract state data for directly assigned slave devices. The target VMM passes these received state data to its MAI to restore the device states of equally assigned slave devices of the migration target VM container. This complete procedure is done, as ensured by our proposed migration strategies, while the execution of the migrated VM is stopped and the slave device state is stabilized by a foregone drain phase.

3.1.1 Live Migration for HPC IaaS

In the following we give an overview of our approach to enable migration in HPC IaaS with directly assigned interconnect adapters. As discussed in Section 2.1 on page 5, HPC IaaS is deployed with physical machines running a VMM for HPC node virtualization. These machines are typically interconnected with each other by using a high performance interconnect network, such as *InfiniBand*, that is also virtualized. We focus our approach on *InfiniBand* in the following.

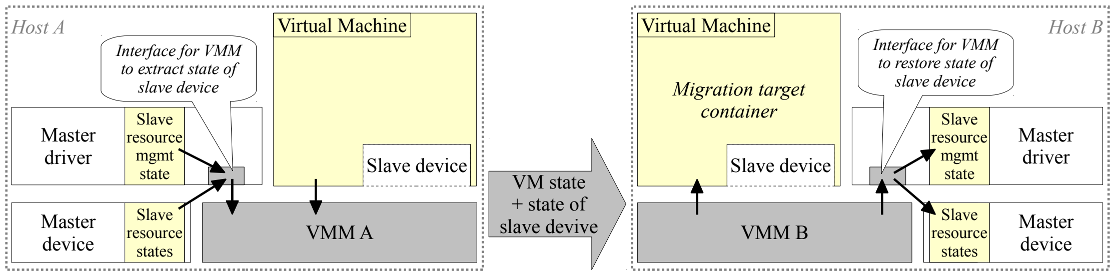
Host A
Interface for VMM to extract state of slave device
Virtual Machine
Master driver
Slave resource mgmt state
Slave device
Master device
Slave resource states
VMM A
VM state + state of slave devive
Virtual Machine
Migration target container
Slave device
Host B
Interface for VMM to restore state of slave device
Slave resource mgmt state
Master driver
VMM B
Slave resource states
Master device

Figure 3.1: Illustration of VM migration by migrating a slave device state by utilizing MAI of the dependent master drivers

To guarantee transparent migration, a HPC VM instance configured with an *InfiniBand* slave device has to be reachable under the same endnode addresses after its migration. Because of this, such a migration in *InfiniBand* can only be performed if the migration target is connected to the same *InfiniBand* subnet. Endnode addresses (LID, slave device GUIDs), and port configurations (e.g. PKeys) are therefore moved with the according VM that is migrated. During migration, the source host releases the assigned resources and identifiers, so that the destination host can adept the settings to its host channel adapter (HCA) and without causing any collisions. Guest-transparency is achieved by reallocating the equal hardware resources for the target slave device while keeping the identifier identical.

As illustrated in Figure 3.2 on the next page, the migration process is coordinated with a global migration strategy that involves suspending of all HPC VMs of the according virtual HPC cluster where a migration happens. This way, *InfiniBand* reconfiguration due to migration can be performed in the VMM layer. Since all VMs are suspended, the reconfiguration process does not affect any running VMs.

A central cloud management instance instructs and coordinates the whole migration process. It has therefore a connection to each VMM of the HPC IaaS data center. These connections are established on a dedicated network which is separated from virtual HPC interconnect, for instance a dedicated *Ethernet* network. Also the migration data is transported over this management network.

To be more specific, in our scenario, each VMM host is equipped with one or multiple *InfiniBand* interconnect adapters that supports self-virtualization with the shared port model (review Section 2.3.5 on page 25). This model is more scalable and has lower potential performance degradation compared to the virtual switch model. In context of live migration, we also benefit from the management of slave devices in the master driver. Each HPC VM node is using a physical *InfiniBand* HCA exclusively while it still gets only a slave device assigned. This way the master device is therefore only responsible for slave device management (review Section 2.3.5 on page 25) and *InfiniBand* resources are not shared anymore. Resource competition and noise is avoided which affects positively on communication-intensive HPC jobs (compare with Section 2.1.2 on page 10). In terms of namespace and endnode address migratability, it is possible to move addresses and identifiers with the device state in this

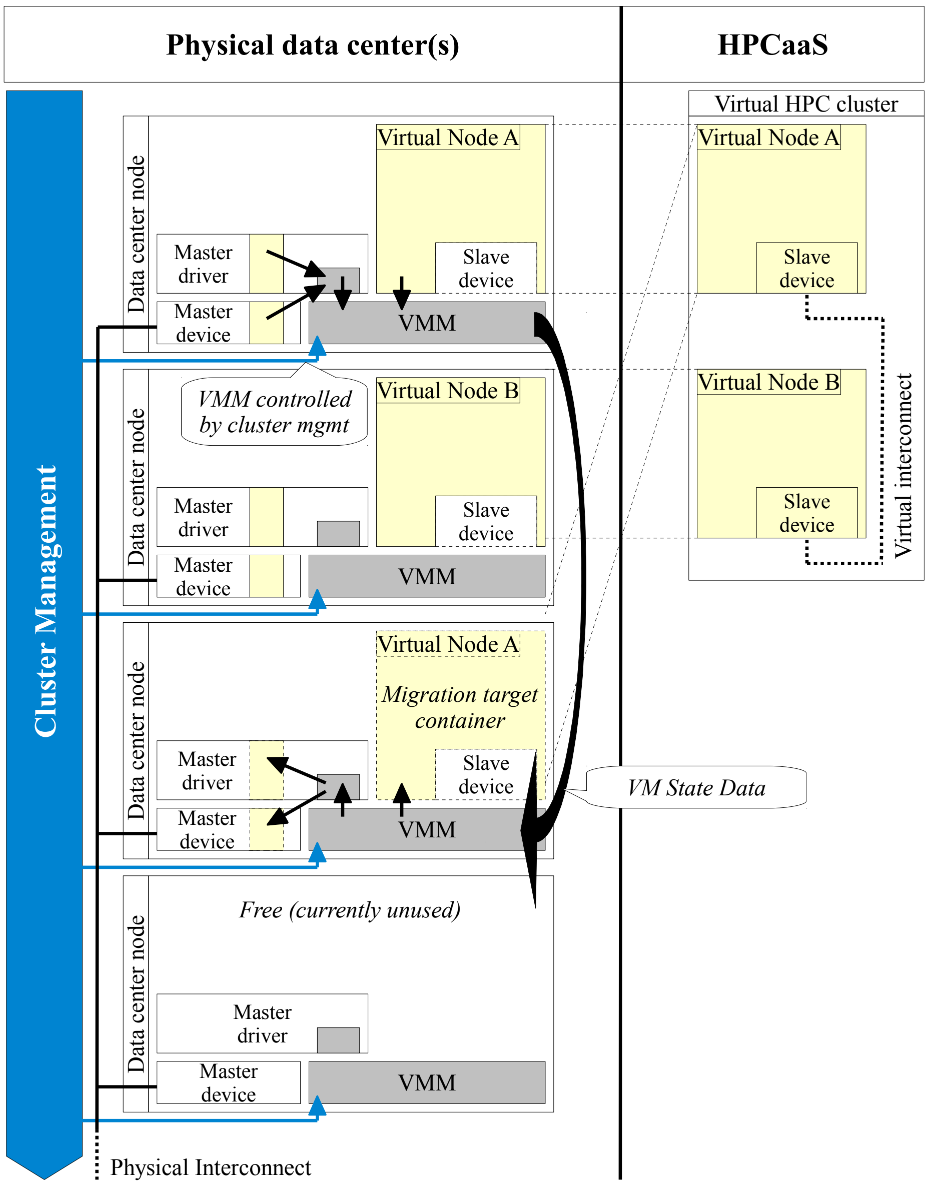

Figure 3.2: Illustration of a live migration in HPC IaaS: Virtual node A is migrated to a different data center node but appears unchanged in the virtual cluster

approach. There are no remaining resources which cannot be migrated because they are not shared. There are also no collisions of identifiers on the destination HCA while migrating (e.g., queue pair numbers (QPNs)). However, this consequently implies that a live migration is only be performable to a target VMM host that has a physical HCA available whose slave device is not in use by another VM instance.

3.2 Migration Strategies

Based on our analysis on Chapter 2 on page 5, we propose two different approaches to realize a transparent migration strategies classified on their degree of involvement. Therefore, we distinguish between local and global migration strategies. We consider a strategy as local, whenever a VM migration is performed without further involvement of remote parties in the migration process. All necessary interconnect adaption is transparently induced by both VMM hosts that are involved in the migration. In contrast to that, a global migration strategy requires further involvements on remote parties in the reconfiguration process during a migration.

3.2.1 Local Migration Strategies

In terms of local strategies, we refer to traditional *stop-and-copy* approaches and their counterparts with reduced service downtime, as introduced in Section 2.2.2 on page 17. In doing so, we propose modifications at the common stop-and-copy stage and the activation stage. As a remainder, the common five-stage model consists of the following stage sequence: (1) Pre-Migration, (2) reservation, (3) stop-and-copy, (4) commitment, and (5) activation.

Modified stop-and-copy stage:
The source host stops the VM execution and also suspends the devices assigned to the VM. After a short drain phase that waits for device quiescence, the host transfers the (preliminary/remaining) VM state together with the device state, including memory pages shared by DMA, to the target host. At the end of this stage the VM and device states are cloned and the execution of both instances is stopped. Also the devices assigned to the VM copy are still suspended.

Modified activation stage:
The target host resumes VM execution and operation of the devices assigned to the VM.

In doing so, we identify the following requirements for the hardware that is migrated. These requirements ensure that a migration can be proceeded transparent for interacting remote parties.

- On interconnect networks; hardware end point addresses are migratable with the VM state.

- On interconnect networks; redirection of communication to the new physical migration destination is performable in a way that it is transparent to all interacting remote parties.
- On interconnect networks; temporary service downtime during the physical host switch can be handled in a transparent to remote parties so that established communication connections are not closed.
- The pass-through device, whose state is migrated, has to be quiet so that state stability during state extraction is ensured. The transfer in a quiescent device state, that we also call device suspension, is done in a transparent way to all interacting remote parties.

Some of these requirements may be already fulfilled by the hardware design and used technology and do not need any further implementation. In typical TCP/IP networks, for example, it is possible to migrate hardware end point addresses, called MAC addresses, with the VM instance. This possibility also performs a redirection of ongoing communication to the new physical target automatically. Also communication data loss associated with device suspension is handled transparently by the TCP/IP protocol as long as the VM service downtime due to migration does not hit any timeout mechanism.

However, if at least one of these requirements is not realizable, due to the design of the assigned hardware and the used technology (e.g., network protocol), a migration cannot be implemented transparent for interacting remote parties. In such a case, we refer to a global migration strategy instead.

3.2.2 Global Migration Strategy for HPC IaaS

For HPC IaaS, we propose a global migration strategy with a four-phase model based on the idea of Scarpazza et al. *Checkpoint-restart* approach [71]. In contrast, our model is still guest-transparent and even transparent to interacting remote parties (compare with Section 5.2 on page 63): While we also suspend the execution of all VMs of the virtual cluster where a migration happens, we move device reinitialization of assigned slave devices complete to the VMM layer by utilizing MAI. Furthermore, and as discussed in Section 2.2.2 on page 17, HPC jobs are typically isolated from third party communication from outside of the cluster. Communication is used only within the cluster for synchronization and intermediate stage coordination purpose, so that we assume that no new communication is initiated on the virtual interconnect of the suspended cluster. Prior the actual migration, a special drain phase also ensures that the virtual interconnect is drained and becomes quiescent. Consequently, the whole migration process is done in a phase of global silence where the state of the according virtual cluster is stabilized. We suggest that this global migration strategy is managed by a migration coordination instance that is part of the cluster management software, such as OpenNebula [50] or Eucalyptus [12].

In detail, this coordination instance transfers the virtual cluster through the following four phases to perform a VM migration (compare with Figure 3.3 on the next page):

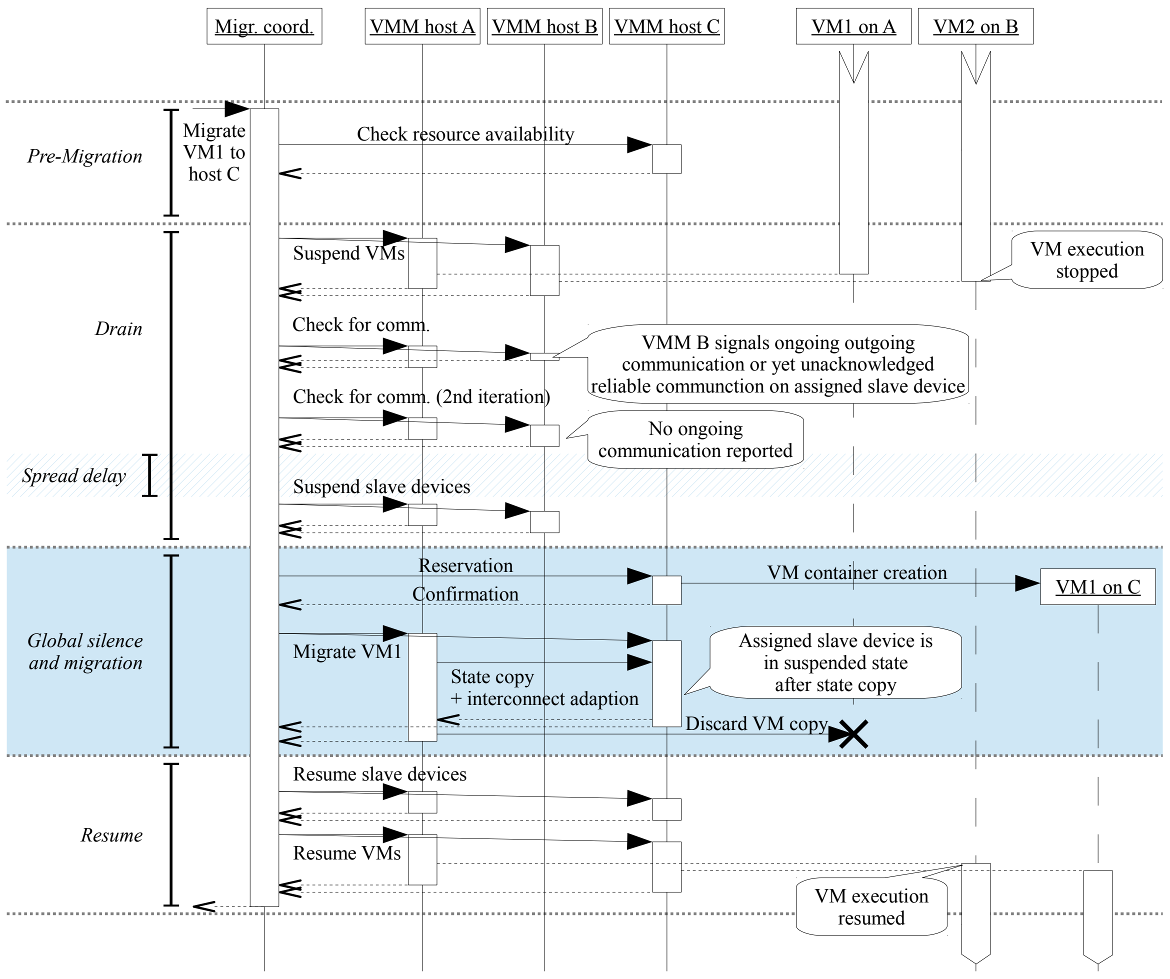
Migr. coord.
VMM host A
VMM host B
VMM host C
VM1 on A
VM2 on B
Pre-Migration
Migrate VM1 to host C
Check resource availability
Drain
Suspend VMs
VM execution stopped
Check for comm.
VMM B signals ongoing outgoing communication or yet unacknowledged reliable communction on assigned slave device
Check for comm. (2nd iteration)
No ongoing communication reported
Spread delay
Suspend slave devices
Global silence and migration
Reservation
Confirmation
VM container creation
VM1 on C
Migrate VM1
State copy + interconnect adaption
Assigned slave device is in suspended state after state copy
Discard VM copy
Resume
Resume slave devices
Resume VMs
VM execution resumed

Figure 3.3: Illustrative sequence diagram of a live migration using our global migration strategy for HPC IaaS: Migrating VM1 from host A to host C

Phase 1: Pre-Migration

At first, target hosts are selected that guarantee resource availability of host-local resources that are equal to the host-local resources currently connected to the preselected VMs that will be migrated. Also a migration sequence may be determined by the coordination instance which defines in which order multiple VMs can be migrated.

Phase 2: Drain

At the beginning of the drain phase, the coordination instance instructs every HPC VMM involved in the virtual cluster to suspend their VMs which are part of that cluster. Afterwards, the drain phase waits until all remaining communication on hardware buffers of the assigned slave devices (review Section 2.1.4 on page 14) are completed to ensure device quiescence. For this purpose, the migration coordination instance queries each VMM involved in the virtual cluster for ongoing outgoing communication on its slave devices assigned to VMs of that cluster. This procedure is repeated until no ongoing outgoing communications and no outstanding reception acknowledgements from remote parties (on hardware implemented reliable communications) are reported by each VMM. That means that every remaining communication was sent out by the assigned slave devices. Then, the migration coordination interface stays a further *spread delay*. This delay should give all remaining unreliable communications the chance to spread out on the virtual interconnect, so that in the end all communication could be received by the slave devices. At last, every VMM involved in the virtual cluster is instructed to suspend the slave devices assigned to VMs of that cluster.

If the actual spread delay, which depends highly on the virtual interconnect itself, guarantees complete drain of communication on the virtual interconnect, the devices assigned to the VMs do not have to be suspended afterwards. However, suspension in this way is only optional as long as the hardware state of the slave devices is quiescent and stable, when there is no traffic on the virtual interconnect.

Phase 3: Global silence and migration

In this phase, the migration processes are made. For each VM that has to be migrated the following three-stage model performs the actual migration:

Stage 1: Reservation

The according target host creates a container for migrating the VM.

Stage 2: Copy

The according source host is instructed to copy the VM state together with the device states to the target host. In this step also the necessary interconnect reconfiguration processes are taking place. In contrast to our local migration strategy, these processes can cooperate with remote VMMs (e.g., to reestablish connections).

Stage 3: Commitment

The target host indicates that the image was successfully copied and that the virtual interconnect was successfully reconfigured. After that, the source host discards its VM instance that was hold in case of migration failures. In the end, the VM state was copied to the according target VMM and its execution, as well as the assigned slave devices, is still suspended.

Phase 4: Resume

At last, the coordination instance instructs all VMMs involved in the virtual cluster to resume the assigned slave devices assigned to VMs of that cluster, if they were suspended in the Drain phase (2). Then the instance let the VMMs resume the execution of all VMs of the virtual cluster, so that this cluster returns to normal operation.

Our introduced global migration strategy approach is only applicable if all interacting remote parties are under the control of the migration coordination instance. For instance, in a traditional IaaS virtual cluster this is not the case as long as the virtual nodes are providing services for remote parties beyond the cluster itself, such as client computers in the Internet. However, interaction with the remote parties in the Internet is based on TCP/IP protocol stack that gives a virtual view about the underlying network and utilizes connectionless packet delivery and that is designed to handle packet anomalies. Such anomalies are packet loss, out of order delivery, and even packet duplication [54]. Thus, live migration in traditional IaaS virtual clusters utilizes local migration strategies.

3.3 Migration Assistance Interface (MAI)

The actual extraction and restoration of the hardware state of a pass-through device and the adaption of the interconnect is done by utilizing a novel interface that we introduce for both strategy classes. We call it migration assistance interface (MAI) because it assists a VMM in guest-transparent hardware state migration while the execution of the according VM is suspended. This interface is implemented at a master driver of a self-virtualizing device using unequal device exposure, as introduced in Section 2.1.3 on page 12. Such a device exposes a master device and multiple slave devices where the master device is used by its master driver to manage and configure the slave devices. Each slave device owns physical resources for direct data movement and direct access to the shared resource. They are intended to be assigned to a VM. Because we expect, that the master driver is executed in a special privileged VM running drivers (e.g., Dom0 on *Xen*) or even within the VMM itself (e.g., the host Linux in KVM), the VMM is able to access the driver. This way it is getting assistance in migrating a pass-through slave device state. The slave device state, which consists of the actual state of the resources of the slave device itself and the resource management state in the master driver, is extracted and restored by routines in the master driver. Based on this approach, we still benefit from near native performance of device pass-through. In the same time, we break up device state opaqueness. The VMM gets device knowledge in an indirect way, since device knowledge still remains in the driver but becomes utilized with MAI for live migration.

For this purpose, MAI provides generic interface functions per slave device organized in three groups: (1) state stabilization, (2) state migration, and (3) emulation assistance. We intend the state stabilization group to guarantee that the slave device is in a stable and quiescent state for a state migration. The VMM calls methods of this group while the VM, to which the according slave device is assigned, is suspended. In order to transfer the slave device state data between the VMMs involved in a migration process, the state migration group provides functions to extract and load state data. These functions are called while the VM execution is stopped. The VMM on the source host has also to ensure that the slave device, from which the state is extracted, is in a quiescent and stable state before state extraction. For this purpose, the VMM utilizes the previous mentioned state stabilization group. The third group, the emulation assistance group, is additionally required for certain devices: After a guest-transparent migration, some slave device register need special runtime correction mechanism, because their states are not restorable by the master driver in any way (compare with Pan et al. [53]). With the correction mechanisms, their state appears unchanged or at least changed in a way that the driver in the guest can continue normal operation. Since the VMM typically has the ability to trap device register access (e.g., KVM actual intercepts

Group	Method	Description
State stabi-lization	**Resume**	Set slave device in a stable and quiescent state
	Suspend	Return slave device into normal operation
	Check for ongoing comm.	Check for ongoing or queued communication on slave device hardware buffers
State migra-tion	**Extract**	Return slave device state data
	Restore	Restore a slave device state with passed device state data
Emulation assis-tance	**Return register correction layout**	Returns the register correction layout (see Section 3.3.3 on the next page)
	Register access handler *X*	Interface method to handle register access *X* in the master driver

Table 3.1: Overview of MAI interface functions

some configuration registers, such as interrupt configuration, to ensure protection isolation), MAI interfaces provide a method whose return value specifies a correction layout for the depending device. This layout specifies which guest accessible device memory regions, also called device registers, have to be handled by special access handler methods on access by the guest. Such handler methods can be completely implemented in the VMM layer or also be a special handler method of MAI to move a correction algorithm to the master driver.

A short overview of MAI interface functions is provided by Table 3.1. In the following, we describe these groups in detail.

3.3.1 State Stabilization

The state stabilization group consists typically of two functions: *suspend* and *resume*. These functions are required by local migration strategies. In case of our proposed global migration strategy for HPC IaaS these functions are also needed as long as the hardware state of the slave is not automatically quiescent and stable after suspending the virtual cluster (see Section 3.2.2 on page 32). This means basically that ongoing DMA transactions are completed (including RDMA), no interrupts are generated anymore and device register values are stable. Furthermore in this strategy, we propose a *"check for ongoing communication"* method. It returns a Boolean value to the caller which specifies if the according slave device is processing ongoing outgoing communications (on hardware implementing reliable communications, this includes waiting for outstanding reception acknowledgements from remote parties of these communication connections). So, this value reveals if all outgoing communications in slave device buffers were sent by the hardware. When the *suspend* method is called, the related master driver transfers the specified slave device in a state that is in a way quiescent and stable. This method guarantees that this state is achieved by returning of the method call. For instance, an *Ethernet* master driver could program a filter at the virtual switch logic that effects virtually disconnecting from the network of the according slave device (all further incoming network packets are dropped by the resource sharing logic). Since some outgoing device operation may still in-flight, such as ongoing network packet sending, the *suspend* interface call returns after the operation ended and the quiescent state is reached. *Resume* reverts the device settings that were done by *suspend* so that the slave device returns to normal operation. The migration target VMM may also call this *resume* method after a successful migration, since we consider device suspension as a part of the actual device state that is migrated. So, whenever the device was suspended on the source host before, the target device, on which the state is restored, has to be resumed prior the migrated VM continues normal execution. Ideally, a slave device should be in a suspended state whenever the execution of the depending VM is suspended.

However, *suspend* and *resume* methods become optional in our global migration strategy proposed for HPC IaaS if the following two requirements are fulfilled: (1) The *drain delay* (see Section 3.2.2 on page 32) guarantees complete drain of communication on the virtual interconnect. No slave device involved in the virtual cluster receives a message that could cause a hardware state change. (2) Such a slave device is also in a quiet and stable state when no messages from the virtual interconnect are received anymore. In this case, a slave device suspension is redundant so that it can be left out for simplicity.

3.3.2 State Migration

The state migration group offers a method to *extract* slave device state data and a method to *restore* a slave device state by passing the previously extracted state data. These MAI methods are called while the system emulator part of the VMMs iterates through all virtual devices to transfer the (virtual) device states of the VM (illustrated in Figure 3.1 on page 29). In this phase or stage (compare with Section 3.2 on page 31), the VM execution is stopped and the according slave devices are in a quiescent and stable state. The master driver on the migration source actually composes the state data that is in turn decomposed by the master driver on target host. This means that the format of the state data is only created and interpreted by the master drivers during the migration process. The VMMs involved in the migration will handle this data as raw data and transfer them from the migration source to the migration destination host. This way, MAI is a generic interface while keeping detailed device knowledge in the master driver. This state data typical consists of state data of the slave device itself and also some state data in the master driver that is part of slave device management.

3.3.3 Emulation Assistance

We observe several types of guest accessible device memory regions, also named as device registers in the following. Some of them impose further challenges in value restoration because they cannot be restored with a simple value copy by the master driver during a slave device state migration. Such registers, which are basically read-only and write-clear registers, require further runtime corrections after the first migration. A read-only register is typically a register containing a statistical value, such as a network packet counter or a ring buffer element pointer. A write-clear register is often a status indicator register that indicate a specific device status by a set bit (e.g., pending interrupt). Software clears (acknowledges) such a status bit by writing this bit back to the device register.

Our approach proposes correction mechanisms for those registers in the VMM layer because the VMM typically establishes the pass-through for the guest to the actual device registers. For this purpose, the emulation assistance group provides a method that return a standardized correction layout to the VMM and optionally multiple methods to handle certain device-specific register accesses. This correction layout specifies exactly which device register regions have to be intercepted on access initiated by the guest. It also specifies in these cases which handler shall be called. The actual handling routines rely in the master driver and are accessible with methods of the emulation assistance group. Alternatively, some handling routines can also be implemented in the VMM because of performance reasons or reasons of implementation reuse for widespread register types on a large number of different devices (e.g., status indicator registers).

The correction setup is instantiated with the returned information of the register layout getter method of MAI when the device pass-through is established by the VMM. This process normally happens when a VM container is created. Later and during VM execution, each access to a device register that shall be handled is intercepted by the VMM. It calls the according register access handler. Additionally, handler methods can also be utilized to only monitor and protocol accesses to observed registers which may also a useful source of information for a state restoration on certain devices (read Section 3.4).

However, how a register access handler is actually implemented depends on the according register of the device itself. For a common set of register types, such as a read-only counter register and a write-clear status register, we provide an exemplary approach in Section 3.4.

3.4 Hardware State Migration

In the point of view a guest VM execution life cycle, the moments of guest-transparent live migrations happen seamlessly, at any point of time, and are completely unpredictable for the guest. Because of these reasons, the assigned slave device on the migration target has to appear in exactly the same or at least in a compatible state that is not unexpected to the guest software when the execution of the VM continues. We expect that for most devices and their drivers such a compatible device state is a state where all remaining work on the device is suddenly completely processed (e.g., a network adapter sent all packets from its hardware queues). This requirement should guarantee that the guest's slave driver continue normal operation and does not fail. Such a state should be automatically reached by the drain phases of our proposed migration strategies.

Nevertheless, we observe therefore two categories of state that has to be migrated: (1) the actual device state data migrated with MAI and (2) device pass-through configuration data. This device pass-through configuration data is related to the VMM and describes how it is configured and mapped into the guest's virtual machine platform. This includes, for instance, on which guest's virtual interrupt line device interrupts have to be routed, where mappable device register memory regions are mapped into the guest memory, or under which address (e.g., PCI device address) the device actual reachable in the guest. We suggest that the according migration of this kind of state is done by the VMM directly since the VMM is anyhow responsible for setup the pass-through configuration.

This way, the pass-through configuration state is migrated directly before the MAI interface is called to extract the actual device state and after a foregone drain phase of the according migration strategy. We notice that this configuration state also includes the state of pending interrupts that occur during the drain phase when the VM was suspended. They are reinjected to the migrated VM on the destination host, immediately after the VM was resumed. This way, the guest's device driver can still handle the events that occurred during the drain phase.

As discussed in Section 3.3.2 on page 37, the actual state device extraction is handled in the master driver. It thereto goes through the resources according to the slave device state and returns all data that are relevant to reconstruct the state on the migration destination. Since it is impossible to evolve generic *extract* and *restore* methods due to the very specific designs of devices (even devices of the same type), the hardware has to be analyzed in detail. Knowledge about hardware interfaces and their behavior are required because wrong kind of access for the purpose of state extraction and restoration may result in unexpected device behavior. The actual implementation details of the MAI routines may also follow a device dependent migration plan (such as proposed in Section 3.5 on the next page for *InfiniBand* devices) due to device internal dependencies.

The actual device state that is migrated with MAI is likewise separable into three main manifestations: (1) the direct observable device state in form of the state of the interaction interfaces, (2) the hidden internal device state, and in case of self-virtualizing devices implementing unequal device exposure (see Section 2.1.3 on page 12) also (3) the resource management state for the according slave located in the master driver and device. The actual fourth state, the guest's driver software state, is automatically migrated with the VM instance but is completely opaque to the master driver and thus to MAI.

The direct observable device state describes states of device components, for instance special device registers for configuration, that do not have a direct dependency to the internal device state. They can normally be copied or, at least, be emulated after migration when they are not directly writable. For this purpose, Section 3.3.3 on page 37 the emulation assistance group of MAI can be utilized.

The hidden internal device state has mainly to be reproducible by traversing hardware state machines. However, if there are some states which are unreachable by traversing (local as well as global) and there are no appropriate compatible states that could be handled by the guest's driver, the according hardware device seems not to be feasible for guest-transparent migration. This is because there is no chance to restore the previous state. Informing the guest that his view about the device state is outdated would violate the the guest's unawareness of migration.

The resource management state in the master driver should be extractable as easiest, since full access to the master device and master driver is given. The master driver can also be modified in any way to support management migration, since this driver is aware of migration in our approach.

3.5 InfiniBand specific Hardware State Migration

In this section, we provide a brief overview of the *InfiniBand* specific implementation part of slave device migration that is executed with MAI's *restore* and *extract* methods. As described in Section 3.1.1 on page 28, we primarily focus on a transparent migration solution for self-virtualizing *InfiniBand* HCA within a fabric. We further focus it on state migration of such HCAs which utilizes the shared port model. A HCA is exclusively used by maximal only one VM instance while it gets only a save device assigned. This way, the master device is still responsible for slave device management, handling the special queue pairs (QP) QP0 and QP1 (review Section 2.3.5 on page 25). Endnode addresses (LID, slave device GUIDs), port configurations (e.g. PKeys), and allocated resources are getting movable in the physical *InfiniBand* fabric. Because these resources and namespaces are used exclusively, there is no dependency to a local HCA due to sharing anymore.

Moreover, using the shared port model automatically introduces further protection and isolation between multiple virtual HPC clusters running on the same *InfiniBand* fabric. For instance, subnet managers (SMs) that are managing the fabric are protected from virtual clusters because QP0 of a slave device has no function (review Section 2.3.5 on page 25). This way, *InfiniBand* fabric configuration is enforced by the cloud management or provider. Because usage of a HCA is exclusive for a VM, partitioning with PKeys that are bound to physical ports could be used as a simple and efficient way to isolate virtual clusters from each other (compare with [21]). We also expect slightly better *InfiniBand* performance results compared to a similar solution based on virtual switch implementation: When a single VM is using a single HCA exclusively, the average amount of switches per communication chain is reduced. Also the number of seen *InfiniBand* endnodes in a fabric is reduced by half since slave devices are already represented by the master device node.

With our current knowledge (more details in Section 4.2.2 on page 48) we assume that the master driver always keeps record of firmware resources allocated by the slave device. This is required, for instance, whenever a slave device suddenly requests a reset. Then the master driver is responsible to release all resources that were allocated by the slave. Further, slave devices pass firmware command to allocate or release a resource through the communication channel to the master driver (review Section 2.1.3 on page 12). The master driver then decides which commands are directly passed to the hardware and which are simulated because of protection and isolation reasons. Based on this assumption, the master driver is able to extract and reallocate the slave's resources.

In the following, we introduce how several entities are migrated which is also illustrated in Figure 3.4 on the next page. A remarkable property of our *InfiniBand* live migration approach is that the entities are restored in a reverse order to the state extraction. This is because there are several dependencies in the *InfiniBand* HCA design. For instance, working with QPs requires a configured LID and the states of QPs have to extract before the LID is released. In the point of view of the target master driver, migrating *InfiniBand* related state data (1) starts with the endnode address movement, (2) continues with the reallocation of hardware resources for the slave, and (3) finishes with the restoration and configuration of the management resources needed by the master driver. The source master driver returns related migration data in this order.

3.5.1 Endnode Address and Port Configuration Migration

From the point of view of the physical HCAs involved in a migration, the *InfiniBand* endnode relevant addresses are GUIDs of the according slave devices, and LIDs. Port configurations, such as PKeys are bound to a physical port. In our approach, both are released at the source host and reassigned at the destination host. The reassignment is done by the SM of the fabric where a key-based authentication is introduced so that the migration destination can verify itself to the SM.

The moving process (illustrated in Figure 3.4 on the following page) is performed by releasing the LID and slave GUIDs on the migration source before the migration destination reassigns these values to the destination slave device. With an extension to the SM, the source driver requests releasing the LID and PKeys and attaches a generated key to that request. Then, the SM stores these LID and PKeys together with the key for later reassignment and assigns new identifiers to the requesting master driver. These new identifiers ensure that the source HCA can return to a normal operational state after the migration procedure. Furthermore, the master driver generates new GUIDs for the slave and replaces accordingly the GUID alias on the physical port. The previous GUIDs, the released LID, the released PKeys, and the generated key are passed together to the master driver at the migration destination via MAI. There, this master driver requests the SM to assign this LID and these PKeys to its HCA. For this purpose, the master driver passes the received LID and key to the SM. The SM is then able to configure the HCA's physical port with LID and PKeys that were stored during the release request. Additionally, the master driver setups the slave GUIDs on the HCA which were received via MAI

We notice, that by migrating the slave GUIDs, the slave GID gets automatically migrated, because the GID is assembled from the virtual port GUID and the subnet prefix which should be anyway equal between migration source and destination.

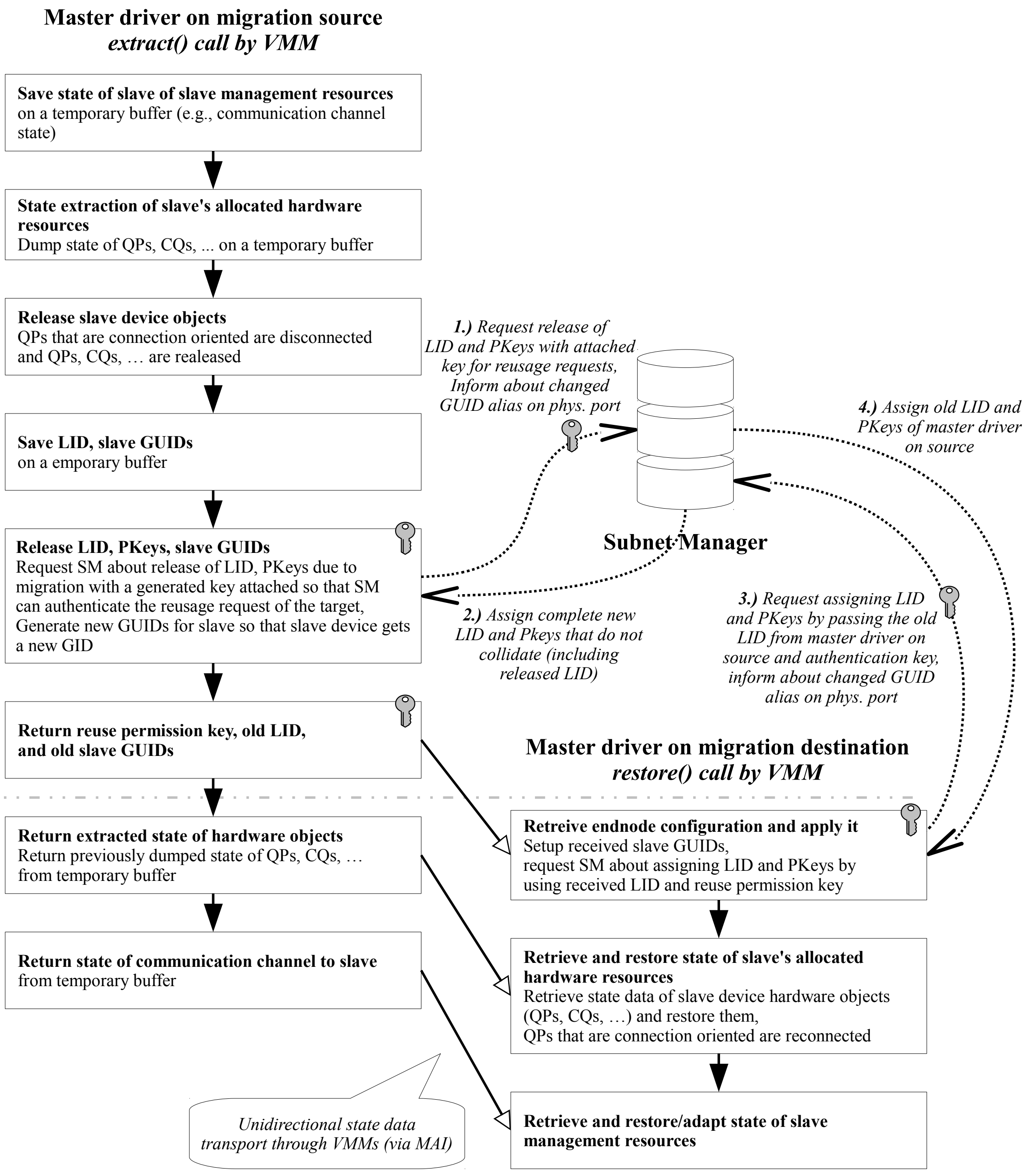
Master driver on migration source
extract() call by VMM
Save state of slave of slave management resources on a temporary buffer (e.g., communication channel state)
State extraction of slave's allocated hardware resources Dump state of QPs, CQs, ... on a temporary buffer
Release slave device objects QPs that are connection oriented are disconnected and QPs, CQs, … are realeased
Save LID, slave GUIDs on a emporary buffer
Release LID, PKeys, slave GUIDs Request SM about release of LID, PKeys due to migration with a generated key attached so that SM can authenticate the reusage request of the target, Generate new GUIDs for slave so that slave device gets a new GID
Return reuse permission key, old LID, and old slave GUIDs
Return extracted state of hardware objects Return previously dumped state of QPs, CQs, … from temporary buffer
Return state of communication channel to slave from temporary buffer
Unidirectional state data transport through VMMs (via MAI)
1.) *Request release of LID and PKeys with attached key for reusage requests, Inform about changed GUID alias on phys. port*
Subnet Manager
2.) *Assign complete new LID and Pkeys that do not collidate (including released LID)*
3.) *Request assigning LID and PKeys by passing the old LID from master driver on source and authentication key, inform about changed GUID alias on phys. port*
4.) *Assign old LID and PKeys of master driver on source*
Master driver on migration destination
restore() call by VMM
Retreive endnode configuration and apply it Setup received slave GUIDs, request SM about assigning LID and PKeys by using received LID and reuse permission key
Retrieve and restore state of slave's allocated hardware resources Retrieve state data of slave device hardware objects (QPs, CQs, …) and restore them, QPs that are connection oriented are reconnected
Retrieve and restore/adapt state of slave management resources

Figure 3.4: Principle of *InfiniBand* slave device migration

3.5.2 Slave Resource State Migration

Hardware resources allocated by the slave (e.g., QPs, completion queues (CQs), memory keys) and also the resources which are necessary for the slave management (e.g., master-slave communication channel) are moved to the destination. However, they are extracted and released before the endnode addresses and endnode configuration (e.g., LID, GID, PKeys) are moved to the destination due to the dependency to these endnode settings. Because of the same reason the restoration is done in reverse order, thus after endnode addresses and endnode configuration has been applied on the target. For this purpose, the master driver on the migration source temporarily buffers the resource states so that it can return the states through MAI in the right order for the migration target (see Figure 3.4 on the preceding page).

Established connections of connection oriented QPs are disconnected at the migration source and reconnected at the migration target. For this purpose, all master drivers of the virtual cluster (except those which are involved in the VM migration) get informed about the migration from the coordination instance. These master drivers can then react accordingly to reconnection requests from the migrated endnode.

4 Prototypical Evaluation

In Chapter 3 on page 27, we proposed a novel design to perform a transparent live migration for High Performance Computing Infrastructure as a Service (HPC IaaS). A migration assistance interface (MAI) is at this the key component of our approach that provides the necessary basic mechanisms to enable hardware state migration. Orthogonal to this, migration strategies were introduced as a way to coordinate the migration process. We proposed a global strategy for HPC IaaS that guarantees migration transparency also to interacting remote nodes of a virtual high performance computing (HPC) cluster. Since the strategies are directly based on MAI, we evaluate the feasibility of MAI, in this chapter, with focus on integrability on Linux KVM and QEMU.

At first, we give a brief overview of our evaluation focus in Section 4.1 and introduce our evaluation platform in Section 4.2 on the following page. In Section 4.3 on page 49, we discuss how we extend QEMU with extra migration procedures to enable migration of pass-through devices and how we integrated MAI. In Section 4.4 on page 57, we briefly introduce, how the MAI interface is added to the master driver in order that QEMU can utilize it. After that, we demonstrate the operability with some testings in Section 4.5 on page 58. Finally, we summarize our evaluation introduced in this chapter in Section 4.6 on page 60.

4.1 Overview

As shown in Section 2.2.2 on page 17, when focused on HPC, cluster node virtualization is typically built with interconnect technologies, such as *InfiniBand*, that effects interacting remote parties on live migration. Because this makes complete transparent live migration impossible, we proposed a global migration strategy for HPC IaaS in Section 3.2.2 on page 32 which introduces a migration coordinating instance to enable at least complete transparent migration at a virtual cluster layer. A therefore needed migration coordination instance is typically anyway implemented in todays IaaS cloud management framework, such as *OpenStack* [52], *OpenNebula* [50], or *Eucalyptus* [12], which needs primarily only to be extended.

However, our proposed strategies are based on the assumption that states of directly assigned hardware can be extracted and restored by the underlying virtual machine monitors (VMMs) without involvement of the guest. For this purpose, we introduced MAI, a software interface at the master driver of self-virtualizing devices (review Section 2.1.3 on page 12). This interface is the key component in our approach of which we show the feasibility and practicability in this chapter.

4.2 Evaluation Platform

We evaluated our approach with *Mellanox ConnectX-3 VPI InfiniBand* host channel adapters (HCAs) [42]. They are one of the first available HCAs on the market that implements self-virtualization in hardware. This self-virtualization feature is activated when the HCA is operated by special driver software. For this purpose, we used an unofficial and prereleased Linux driver[1] provided by *Mellanox Technologies* as well as the according driver source code and necessary HCA firmware[2] [41]. This driver packet is intended by *Mellanox* to be used on a typical Linux Kernel-based Virtual Machine (KVM) virtualization stack. Further, the packet came with the OpenFabrics Enterprise Distribution (OFED) version 1.5.3, an open-source software stack that provides RDMA and kernel-bypass communication interfaces to applications on top of *InfiniBand* or *iWarp* fabrics [49].

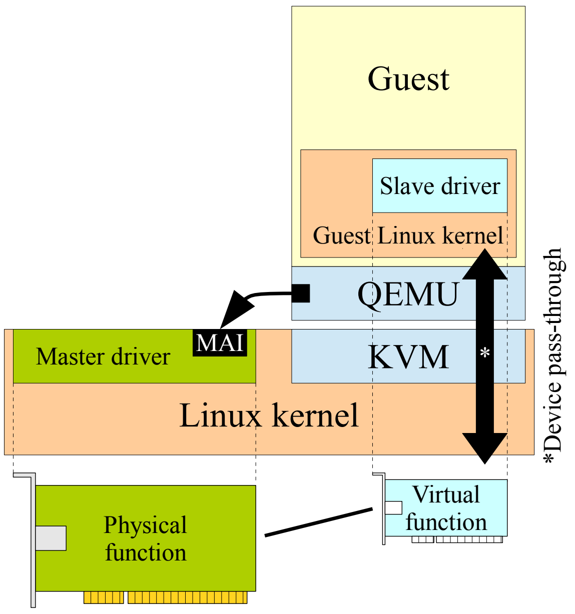

Figure 4.1: Illustration of our evaluation platform

In addition, we used an official Linux kernel [77] for our experimental developments and replaced the *InfiniBand* driver stack with the source code from *Mellanox*. In this context, we tried several different kernel versions, but it turned out that *Mellanox's* driver code, which was originally shipped to be used for a 64 bits Linux kernel of a *Red Hat Enterprise Linux 6.2* (*RHEL 6.2* x86_64) [69] distribution, fits best to the stable kernel version 3.3.8. With it,

[1]Version: SRIOV-ALPHA-3.3.0-2.0.0008 of April, 2012 for *Red Hat Enterprise Linux 6.2* 64 bits (*RHEL 6.2* x86_64). This driver enables the SR-IOV mode of the HCA, a standardized hardware I/O virtualization mode for PCI Express devices.

[2]In the meantime, *Mellanox* published patches [46] for the official Linux *InfiniBand* stack to enhance their public drivers with SR-IOV capability on the Linux RDMA development mailing list [38]. With mainline Linux kernel 3.7.0 they came publicly available. We expect that our modification can be also ported to that code base.

we could replace the kernel's *InfiniBand* stack without further source code adjustments. So we assume that *Mellanox* itself developed their driver on a close kernel release. The kernel components of KVM are included in mainline Linux, as of version 2.6.20 [31]. Accordingly, the KVM version we use comes with Linux kernel 3.3.8. Since the KVM kernel components are only providing a Linux *IOCTL* interface to applications to utilize hardware virtualization features, we used *QEMU-KVM* as the system emulator software. *QEMU-KVM* is a fork of QEMU [4, 66] and is maintained by the KVM development team. This fork mainly adds processor code that uses the KVM interface of the kernel instead of software emulation [32]. Despite the current ongoing code merging process of the *QEMU-KVM* modifications to QEMU project, *QEMU-KVM* still provides better performance [34]. Because of this fact, we use *QEMU-KVM*[3] version 1.0.1 in our prototype.

4.2.1 Virtualization with KVM and QEMU

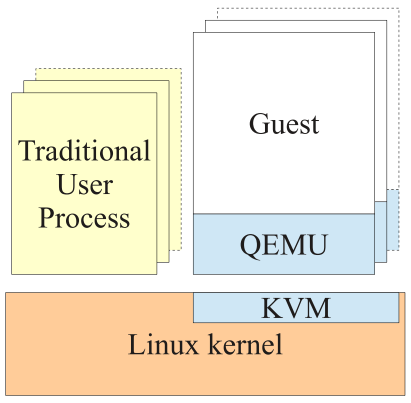

Figure 4.2: Typical KVM system with QEMU (source: [67])

A typical KVM-based virtualization installation consists of the KVM kernel components itself and a system emulator application in user-space, such as QEMU (illustrated in Figure 4.2). KVM itself is a Linux subsystem which adds virtualization capabilities to the Linux kernel and exposes a device node that can be utilized by user space applications to create and run virtual machines [32]. QEMU emulates required system hardware to run unmodified guests while utilizing the KVM kernel interface to execute a virtual machine (VM) in a special guest mode accelerated by the host hardware. In this way, VMs appear as regular Linux processes that integrate seamlessly with the rest of the system [32]. In the end, the KVM implementation benefits from leveraging the large existing feature set of the kernel [32], such as existing process scheduling, memory management, and driver implementations [67]. QEMU processes can be instantiated manually by passing the virtual machine configuration as parameter and can be monitored by regular Linux process management tools (such

[3] In this document, we term the *QEMU-KVM* fork as QEMU.

as the *ps* command of various Unix and Linux systems [65]). Management frameworks, such as *libvirt* [37], create and monitor for each VM a QEMU process in this way.

4.2.2 Mellanox ConnectX-3 VPI

Mellanox's ConnectX-3 VPI [42] is an *InfiniBand* HCA for the PCI Express bus. It implements self-virtualization using unequal device exposure (as introduced in Section 2.1.3 on page 12) and conform to the SR-IOV standard [59]. This standard was proposed by the *PCI-SIG* group [58], a consortium of nameable electronic companies, as extension to the PCI Express standard. It defines which interfaces a self-virtualizing hardware provides and defines that all slave devices appear in the system as a regular PCI Express device. This way a VMM can directly assign a slave device to a VM by using the same pass-through implementation as for traditional PCI/PCI Express devices [82].

The *Mellanox ConnectX-3 VPI* implements resource sharing with the shared port model, as we introduced in Section 2.3.5 on page 25. In this model, multiple slave devices share the physical port of the HCA. Because of this, the *InfiniBand* endnode address (LID) and port configuration (e.g., PKeys) are shared among the master and slave devices. Also name spaces, such as queue pair numbers (QPNs) or memory keys, are shared between all device instances of the physical HCA.

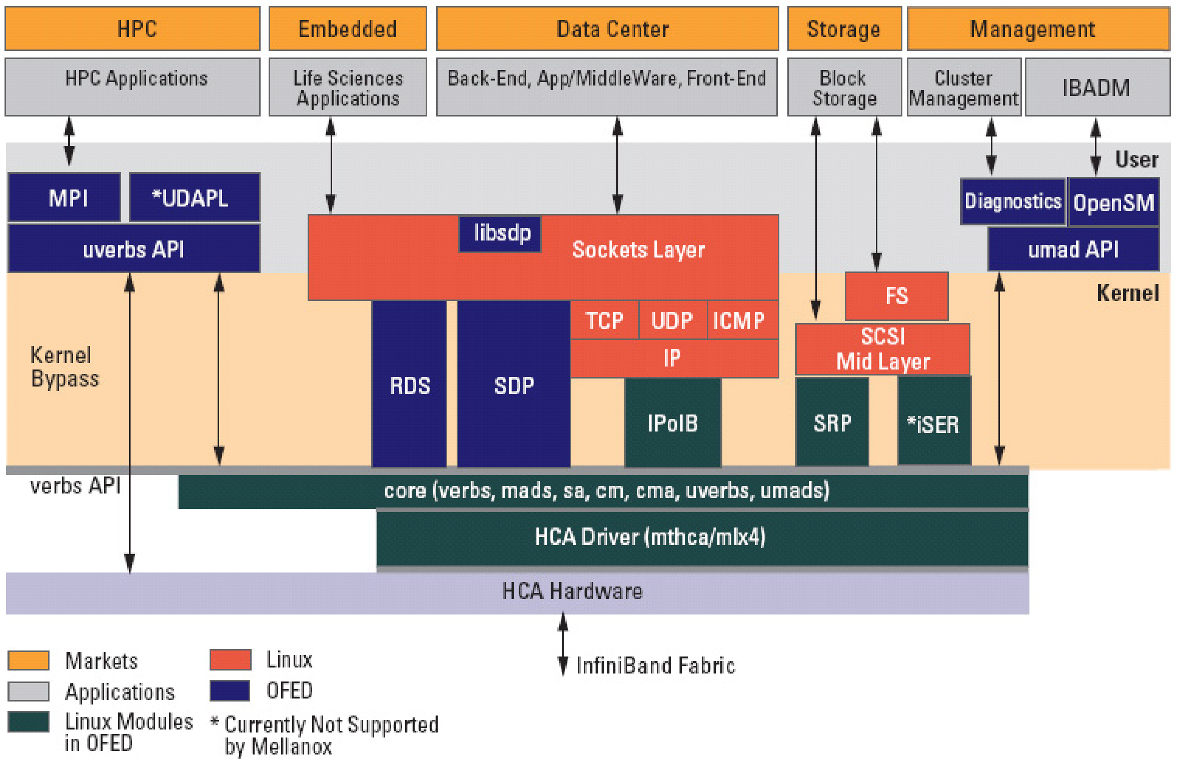

Figure 4.3: *Mellanox* OFED stack (source: [43])

The OFED software stack, which is delivered with *Mellanox's ConnectX-3 VPI* driver, is an open-source software stack that provide OS-bypass interfaces and RDMA communication to applications on top of *InfiniBand* or *iWarp* fabrics [49]. As shown in Figure 4.3 on the preceding page, OFED supports a wide range of upper layer protocols (ULPs), such as IP over InfiniBand (IPoIB), SCSI RDMA Protocol (SRP), on top of the core driver software [43]. This core driver consists of the actual HCA driver module and the so called mid-layer core [43]. This mid-layer, in turn, provides programming interfaces of the HCA hardware for kernel and user space. Especially, the *verb* API of the *InfiniBand* architecture forms the main interface to the message transport service of *InfiniBand*. As also illustrated in Figure 4.3, a typical OFED-conform HPC application is using the *verbs* interface exposed to user space (called *uverbs*) directly or indirectly (for instance with *MPI* [78]).

The OS-bypass interface is therefore implemented on the *Mellanox HCA* as follows: As introduced in Section 2.3 on page 20, an application instructs send and receive to by adding work queue elements to an allocated queue pair (QP). While this data exchange is working via DMA, the application has to inform the HCA about a new element in a QP. In *Mellanox's* implementation, a doorbell register is rung at a write-only device register region mapped into the applications address space. Such a region is called user access region (UAR) whereof multiple UARs exist on the hardware (see Figure 4.4 on the next page). Further, using polling on the completion queue (CQ) to check for finalized HCA operation, removes also signaling the process by the HCA through the kernel. This way, applications can excessively utilize the *InfiniBand* HCA without concerning performance degradation due to too high arrival rates of hardware interrupts that traverse the kernel [45].

As discussed in Section 2.1.3 on page 12, *Mellanox* implements a communication channel between master and slave driver for passing firmware commands from the slave [41]. A slave device has a small set of firmware commands available and writes them to a memory buffer in the guest driver which is shared via DMA with the slave device. When the slave driver writes to a special communication channel register, the master driver gets triggered to process a command requested by the slave. For this purpose the master driver instructs a read command to the HCA which copies the guest's memory buffer contents to the master driver. After that, the master driver simulates a command, modifies it to ensure slave isolation, or passes it directly to the HCA. Finally, the driver let write the command result back to the guest's buffer and instructs the HCA to inject an interrupt at the slave device. This interrupt notifies the slave that the requested command was processed. In general, firmware commands are only used for allocation and removal of hardware resources, such as CQs, QPs, and memory keys (LKeys and RKeys). The rest of interaction is performed with these allocated hardware resources that utilize DMA, interrupts and directly mapped UAR pages.

4.3 MAI Support in QEMU

In this section we introduce our modifications made on QEMU to enable migration of pass-through devices conforming to our approach (sec:prototype:mai-qemu:mods). For this pur-

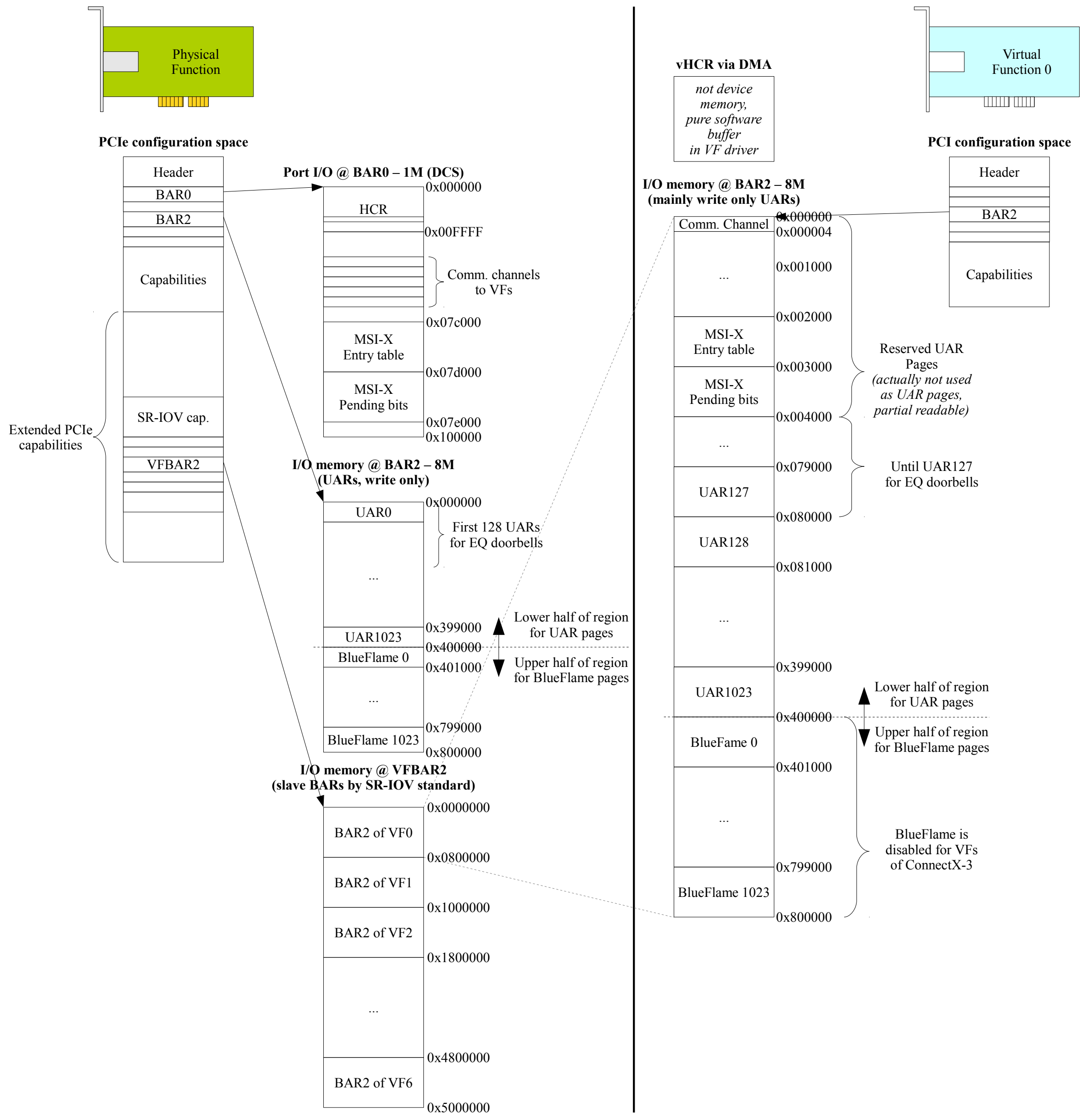

Figure 4.4: Device register layout of *Mellanox's* SR-IOV capable ConnectX-3 HCA (derived from [41, 55, 59])

pose, however, we first provide further background of the already existing virtual device migration framework in QEMU (Section 4.3.1) and how PCIpass-through is implemented (Section 4.3.2).

4.3.1 Guest Device Migration Framework in QEMU

QEMU instantiates a device tree containing objects. Each of them represents a particular device of the VM platform. During snapshot or migration, QEMU goes through this device objects tree and calls consecutively registered state extraction handlers. On each call, an abstract sequential writable data stream object (called `QEMUFile`) is passed to the handlers. This object represents either a VM snapshot file or a communication connection to a remote QEMU process as migration target. Each handler just writes its state data that it is needed for restoration to that stream [64].

The VM device state restoration process is implemented in the reverse way. When restoring from a snapshot file or receiving a migration stream, the state data source is also represented by this abstract stream object, however as a sequential readable data stream, this time. This object is passed to each restoration handler that was registered at an according device tree object. Each handler picks their state data from the stream. The following handlers are doing the same but start with the offset in the stream where the predecessor stopped.

4.3.2 PCI Pass-Through in KVM/QEMU

The KVM and QEMU version we used in our experiments support device pass-through of PCI devices[4]. Such a PCI device provides three general types of interaction [10]: (1) device registers, (2) interrupts, and (3) host memory regions that are shared with the device.

Device registers are accessed by software and form a simple interface to the device. In the PCI standard, device register are subdivided into three types according to the address space where they are appear [57]: (1) the PCI configuration space with a standardized layout (type 0 for PCI endpoints), (2) device register regions accessible via the port I/O instruction interface of the processor [27], and (3) memory mapped device register regions that are directly mapped into the physical address space of a host.

The PCI configuration space (shown in Figure 4.5 on page 53) has the most important role in device configuration. Its layout of registers is standardized and is also used in the device detection process while a host boots up. Their registers are used to detect the device and its capabilities, and to configure the interrupt and device register resources into the system. For instance, so called baseline address registers (BARs) are used to detect and define the respective address locations of additional memory mapped I/O and port I/O regions of the device

[4]PCI Express was not supported yet. However, from the system software point of view, the main difference between PCI and PCI Express devices is the extended configuration space of PCI Express (see [55, 57]). Although the *ConnectX-3* VPI HCA is a PCI Express device, its slave devices appear as fully compatible PCI devices that do not use the extended configuration space.

Interaction type	PCI interfaces
Device register	PCI configuration space, Memory mapped I/O, Port I/O
Interrupts	Interrupt pins (INT# lines), Message signaled interrupts (MSI and MSI-X)
Shared memory	DMA (including RDMA)

Table 4.1: Overview of interaction interfaces of a PCI device

[57]. A unidirectional linked list of capabilities is located at the PCI capability space. Such a capability has its own defined layout and extends the PCI device with further functionality. The address of the first capability of this list is defined with a particular capabilities pointer register [57]. The layout of memory mapped and port I/O register regions is basically completely defined by the vendor. An exception is the interrupt configuration via MSI and MSI-X (configured via a PCI capability). Because this PCI standard supports multiple interrupts per device via PCI bus messages, each of the interrupt is configured through a MSI/MSI-X entry table. The capability structure defines, for instance, in which additional register region the table is located. It specifies the index to the according BAR (see Figure 4.7 on page 55).

Interrupts can be generated by the hardware and notify an event to the software. They interrupt the current program flow of the processor (as long as it is not masked) to let it execute a special service routine. This routine then handles the arrived device event and returns to the normal program flow afterwards. The PCI standard introduces two different interfaces of generating and passing interrupts to the processor: Interrupt pins (typically shared among other devices) and message signaled interrupts (exclusive usage; MSI and MSI-X).

Host memory that is shared with a device is accessed by the device independently and happens also concurrently to the program flow by using direct memory access (DMA). DMA transactions are typically controlled by a DMA entity on the device and are configured with a device-specific register interface.

PCI device pass-through is mainly handled by KVM but established by QEMU. In general, device assignment is achieved by registering or utilizing interaction handlers that connects the according physical device interfaces provided by Linux with the device interfaces presented to the virtual machine platform (see Figure 4.6 on page 54). From the point of view of the guest, a directly assigned device integrates seamlessly with the rest of the virtual system. Because the physical device may appear with a different address and configuration in the host system as in the guest system, QEMU and KVM are translating device accesses by the guest and vice versa.

Figure 4.6 on page 54 illustrates in detail how QEMU establishes device pass-through. Since the assigned device still appears regularly in the host Linux, QEMU has to ensure that an

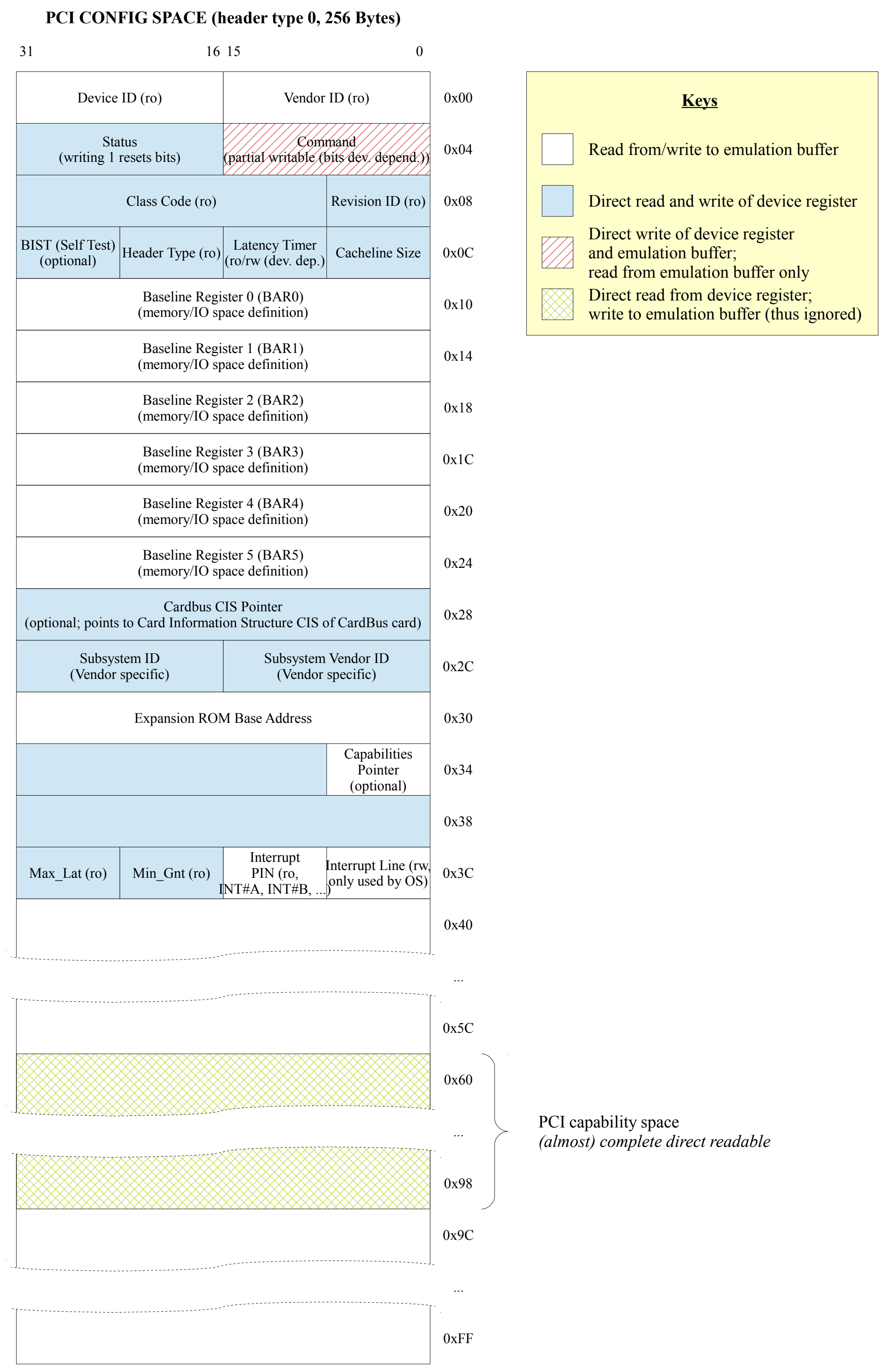
PCI CONFIG SPACE (header type 0, 256 Bytes)
31
16 15
0
Device ID (ro)
Vendor ID (ro)
0x00
Status (writing 1 resets bits)
Command (partial writable (bits dev. depend.))
0x04
Class Code (ro)
Revision ID (ro)
0x08
BIST (Self Test) (optional)
Header Type (ro)
Latency Timer (ro/rw (dev. dep.)
Cacheline Size
0x0C
Baseline Register 0 (BAR0) (memory/IO space definition)
0x10
Baseline Register 1 (BAR1) (memory/IO space definition)
0x14
Baseline Register 2 (BAR2) (memory/IO space definition)
0x18
Baseline Register 3 (BAR3) (memory/IO space definition)
0x1C
Baseline Register 4 (BAR4) (memory/IO space definition)
0x20
Baseline Register 5 (BAR5) (memory/IO space definition)
0x24
Cardbus CIS Pointer (optional; points to Card Information Structure CIS of CardBus card)
0x28
Subsystem ID (Vendor specific)
Subsystem Vendor ID (Vendor specific)
0x2C
Expansion ROM Base Address
0x30
Capabilities Pointer (optional)
0x34
0x38
Max_Lat (ro)
Min_Gnt (ro)
Interrupt PIN (ro, INT#A, INT#B, ...)
Interrupt Line (rw, only used by OS)
0x3C
0x40
...
0x5C
0x60
...
0x98
0x9C
...
0xFF
PCI capability space
(almost) complete direct readable
Keys
Read from/write to emulation buffer
Direct read and write of device register
Direct write of device register and emulation buffer; read from emulation buffer only
Direct read from device register; write to emulation buffer (thus ignored)

Figure 4.5: Pass-through setup in QEMU of PCI config space (header type 0)

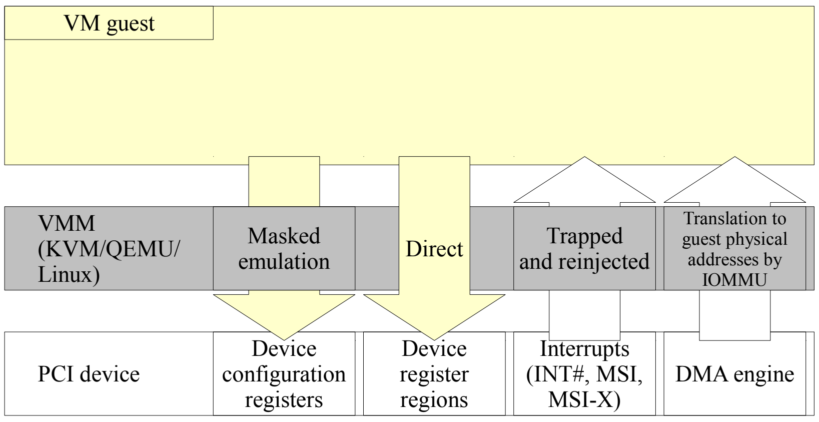

Figure 4.6: Ways of device interaction and how QEMU handles pass-through

untrusted guest cannot configure the device in a way which would violate protection and isolation of the guest.

Configuration registers are mainly emulated by QEMU. Access from the guest is only for a few registered passed to the real hardware. The BARs, for instance, are completely emulated, because modifications would change the location of I/O register regions in the host system. QEMU actually reads the values from emulated registers and relocate the mappings only in the guest memory. The QEMU configuration register emulation is thereto implemented by an overlayed buffer, a write mask and a read mask. Each write access is written to that buffer and also to the device as long as the requested register was not masked out. Each read access is read from the device when the requested register was not masked out. Otherwise, the value is read from the emulation buffer. Masking can be set bitwise and the masks are initialized when QEMU initializes the depending pass-through device object.

Access to I/O memory regions (port and memory mappable) are directly passed to the device. The memory mapped regions are therefore directly mapped into the guest memory space and port I/O regions are trapped by QEMU. However, QEMU passes here the port I/O request directly to the Linux's port I/O interfaces [82]. The MSI and MSI-X configuration tables that rely in one of the device's I/O regions form also here an exception:. QEMU overlays the according regions with extra emulation buffers. Each time, the guest modifies one of these buffers, QEMU instructs KVM to resetup the interrupt routing accordingly.

Interrupts from the device are routed through KVM. KVM registers therefore for each interrupt its own interrupt handler at the Linux host and configures the device accordingly. When an interrupt arrives, KVM receives the interrupt and reinjects it to the guest according to the guest's view of the interrupt configuration. Every time when the interrupt configuration is changed by the guest, QEMU instructs KVM to resetup the routing. This way, the hardware register will always contain a interrupt configuration, that fits in the host configuration while the guest gets only a emulated view about the interrupt configuration.

DMA access from the device is protected with a special virtualization hardware, the IOMMU

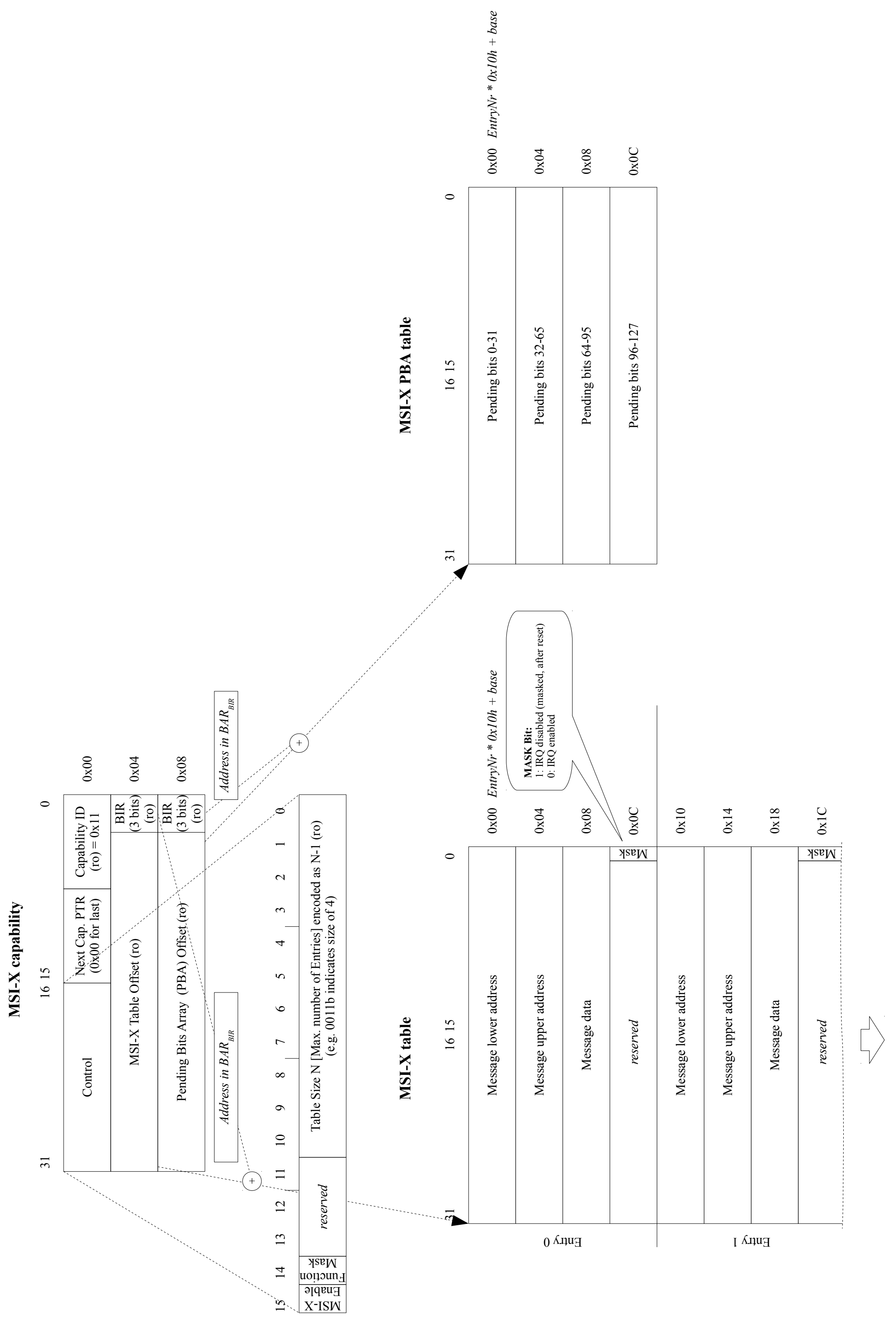

Figure 4.7: MSI-X PCI capability layout (for MSI table layout or more information read [56])

[82]. It is part of the hardware virtualization features and extend recent computer platforms. Such a implementation comes with *Intel's VT-d* or *AMD's IOMMU*. The IOMMU works like a typical MMU but translates addresses for devices instead of processes. In KVM, the IOMMU is configured to translate each memory access by DMA for the pass-through device into an access to a guest memory addresses. KVM pins therefore the whole guest memory, since swapping is not supported yet [82]. In the point of view of the assigned device, the guest memory address space appears completely as the host memory address space.

4.3.3 Extending QEMU's pass-through code

We extended QEMU's pass-through code with an extract and restore methods and generally unlocked migration of pass-through devices. These methods are migrating the device configuration state and pass the state data of the according MAIs through QEMU's migration stream. For this purpose, we implemented the following functionality to the restore and extract functions:

First, the functions copy the PCI configuration space emulation buffer from the source to the destination, since QEMU's device configuration emulation implementation put each write access also to the emulation layer. Except of the status register, each register value is also written back to device when writing was not masked out. Because the status register implement a Write-To-Clear semantic, each bit that is written to the register will reset the according bit. Due to this fact, we added an additional emulation buffer for this register (see Figure 4.8 and Figure 4.9 on the next page): During migration the actual status bits from the device or'ed with the value of the emulation buffer on the source host is copied into the emulation buffer at the migration destination. Whenever the guests accesses this particular register, QEMU returns instead of the original device register the register value on the device or'ed with the value of the emulation buffer. On write access, the Write-To-Clear semantic is simulated on the emulation buffer while the request is also forwarded to the device. Further, the restore and extract functions copy the contents of the overlayed MSI/MSI-X table buffers to the destination, if there are any (see Figure 4.7 on the preceding page).

After this first migration process, the restore function calls all internal QEMU routines that are also called when the guest changes a value of an emulated configuration register. This way, QEMU reestablishes the interrupt routing setup and maps device register regions accordingly to the guest memory.

Finally, the functions call the according MAI methods so that the actual hardware state gets also migrated (described in Section 4.4 on the next page).

The IOMMU setup is automatically established when QEMU instantiates the pass-through device object. There is no data for the IOMMU configuration setup that needs to be exchanged to the destination. Because the mapping of guest memory pages to physical host pages may be different on the target host, the IOMMU would contain different entries but with the same effect in the point of the view of the guest an the assigned device. In their view the layout has not changed. Further, guest memory pages shared with the device are already migrated with QEMU's iterative memory copy routines. As we discussed in Section 2.2.2 on page 17, the device should be in a quiescent state before QEMU does this.

I/O memory regions are not migrated within QEMU directly since this is a task of the master driver with device knowledge. However, pending interrupts are not migrated yet, since we did not need them for our tests. Pending interrupts are occurring when the VM execution was stopped while the pass-through device still generates interrupts. We will do this in a future work.

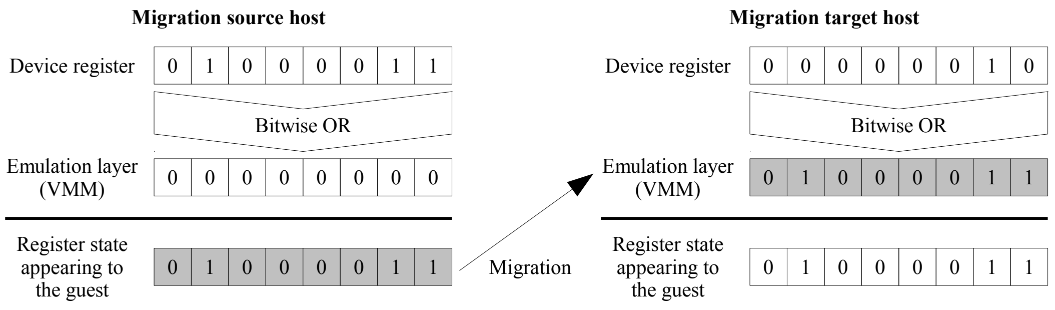

Figure 4.8: Migration of the status register in the PCI configuration space

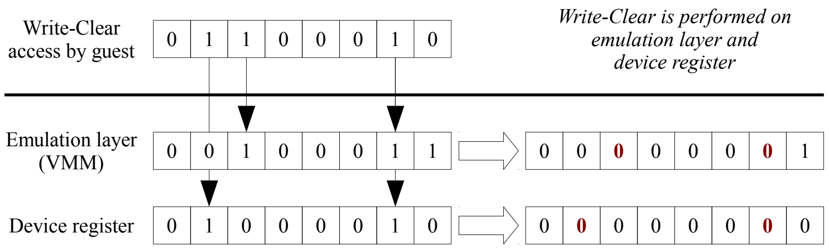

Figure 4.9: Write access from the guest on the status register

4.4 MAI Integration in Master Driver

We implemented the actual MAI at the master driver with a Linux IOCTL device file. The path of this file is passed as parameter to the pass-through configuration of the QEMU instances. Each MAI method is encoded with an own IOCTL identifier (review Section 3.3 on page 35) and gets accordingly called by the novel restore and extract methods of QEMU's device pass-through implementation (review Section 4.3.3 on the facing page). Further we enforced the master driver to enable only a single slave device, as required by our *InfiniBand* specific design (review Section 3.5 on page 40).

In the first place, we only implemented the *extract* and *restore* methods for our evaluation. They look as the following:

The *extract* interface gets a pointer to a user-space buffer passed to where the master driver writes its state data into. However, since the data length depends on the actual device state (e.g. number of allocated QPs), we introduced a second interface which returns the expected size of state data in bytes. This value matches the actual size, when the interface is called after the device has reached its quiescent state. During this state the state data will not chance.

The *restore* interface gets a gets a pointer to a user-space buffer passed from which the master driver is reading the state data directly.

This means in the end, that QEMU's migration implementation needs to temporarily cache the device state data on buffers. We extended this to our restore and extract functions within QEMU (see Section 4.3.3 on page 56) so that the following happen whenever one of the two MAI methods gets called by QEMU: On the migration source, QEMU first requests the expected state data size from MAI and then allocates a buffer accordingly. This buffer is then passed to MAI's *extract* method. After the IOCTL finished, QEMU puts first the size of the buffer and then the buffer contents itself to the migration stream. On the migration target, this size info is used to allocate a buffer where QEMU puts the incoming state data from the stream into. This way, QEMU calls *restore* of MAI, after the device state data arrived completely.

4.5 Evaluating MAI Integraton

Up to now, we integrated MAI into QEMU (see Section 4.4 on the previous page) and introduced code for migrating the pass-through configuration (see Section 4.3 on page 49). In this section we evaluate the functionality of these changes. For our tests we use two identical hosts[5] that are interconnected with each other through *Ethernet* and a *InfiniBand* interconnect with a SM running on a third party. On these systems, as well within the VM, we installed *CentOS* 6.2 *x86_64*, a *RHEL 6.2 x86_64* compatible Linux distribution [5], and *Mellanox's* OFED stack including the HCA driver [41] (review Section 4.2 on page 46). On the physical hosts, we use Linux kernel 3.3.8 with our modifications to the HCA driver (see Section 4.4 on the preceding page). and the patched QEMU 1.0.1 (see Section 4.3.3 on page 56). The VM is running a unmodified guest system. During migration the VM state is migrated through the separate *Ethernet* network.

To show that pass-through configuration migration work, we check the state of the configuration space within the guests, and list the interrupt handler registered by KVM at the VMM hosts before and after a migration (`/proc/interrupts`). We also run a small tool from the OFED stack [41], called `ibstat`. This tool communicates with the slave device driver, which therefore utilizes the slave device to extract some device properties (e.g., HCA summary, port link state, GUIDs). The slave driver communicate therefore with the master driver by passing firmware commands through the communication channel. The master driver ac-

[5]Intel Xeon E3-1230, 16 GiB DDR3 ECC main memory, Onboard Gigabit *Ethernet* adapter, *Mellanox ConnectX-3 VPI InfiniBand* HCA PCI Express

tually passes the commands to the HCA and returns the results to the slave device. This communication involves access to slave device registers in an I/O memory region to establish this channel, DMA when the master driver reads the according firmware command and its arguments from a software buffer within the guests slave driver, and interrupts when the master driver informs the slave driver that the command was processed and its result was written (via DMA) back to this software buffer.

To perform the tests, we first start a guest instance on the first machine. After it booted up, we run `ibstat` and then unload the HCA driver in the guest. We also print the state of the guest visible configuration space via the Linux sysfs interface (`/sys/bus/pci/devices/****:**:**.*/config`).

After that, we migrate the VM to the second host where we already instantiated a QEMU container process to receive the VM state. The migration is performed through the *Ethernet* connection and via the `migrate` command of the QEMU internal command line interface, called QEMU monitor.

At the destination, we first compare the guest visible configuration space with its state before the migration. We observe, that it is identical, even the status register bits appear equal. Then, we reload the HCA driver, so that the new pass-through device gets initialized. However, we repeat unloading and reloading the driver several times since unexpected values were returned from the communication channel. After two or three times, the driver comes up. We are currently not sure what exactly the issue is at this point, but we believe that it has something to do with the slave management state of the master driver that is not migrated yet. After the driver comes up, we start `ibstat` and compare the output with the previous one: We observe that HCA dependent values, such as GUIDs, are changed in the output. This proofs us, that the firmware commands could be processed by the master driver at the destination and that the communication with this driver is working.

However, interrupt routing configuration migration could not be completely tested since we had to unload the driver at the migration source. This disables interrupts at the device which are only re-enabled when the driver is loaded again. When we migrate a VM without unloading the driver we observe that at the point of adaption of the pass-through configuration, the interrupt routing gets reestablished by the KVM module at the destination. But because the slave interface of the HCA was never initialized, we could not trigger any interrupts to confirm that this setup is actually working. Even so, the observable registered handler configuration (interrupt number to handler mapping) appears equal to the one that is established when the guest driver comes up.

Further, we implemented returning a test string in the MAI's *extract* interface. This data gets transported on each migration which we observe at the migration destination: The MAI's *restore* interface is implemented to print the received test string to the Linux kernel output, which it actually does.

4.6 Summary

We implemented the MAI interface at the master driver and integrated its usage in QEMU. For this purpose, we extended QEMU to support migration of pass-through devices and implemented state extract and restore routines. They first migrate the pass-through configuration within QEMU and then the hardware state data originated from MAI's methods.

In a test scenario we demonstrated that MAI integrates well in QEMU, migration of PCI pass-through configuration state works with the *Mellanox ConnectX-3 VPI InfiniBand* HCA, and state data can be transported via MAI from the master driver at the migration source to the master driver at the destination. This way, MAI has the potential to extend QEMU indirectly with device knowledge.

5 Related Work

Several recent studies have introduced methods and concepts that enable live migration of virtual machines using a pass-through device. However, as we observed in our research, there is no approach known to us that focuses on guest-transparency. Because early studies of pass-through device migration were mainly focused on *Ethernet* network adapters we first introduce their proposed techniques in Section 5.1. We point out the differences to modern interconnect networks, represented by *InfiniBand*. After that, we introduce recent works in live migration of *InfiniBand* adapters in Section 5.2 on page 63.

5.1 Ethernet-based Approaches

Approaches of live migrating directly assigned *Ethernet* devices, typically utilize implicit assumptions of common TCP/IP networks. These assumptions fulfill our proposed requirements for local migration strategies (see Section 3.2.1 on page 31): End point addresses are migratable with the virtual machine (VM) instance, ongoing communication can be transparently redirected to the new destination and network service downtime, as well as handling of communication anomalies that may occur (e.g., package drops and duplication) is part of the TCP/IP protocol stack. This means that a migration of a VM is transparent to communicating remote parties as long as the VM service downtime does not hit any communication timeouts. This way, the primary focus of these studies relies on reducing the total VM service downtime caused by hardware reinitialization.

A common approach utilizes a kind of hotplug mechanism on the assigned device which notifies the migrated guest about a replaced device [3, 30, 74, 83]. The device is hot unplugged before migrating and a new device is plugged after a completed migration on the target host. However, Kadav et al. [30] showed that hot device unplugging and plugging can cause a considerable service downtime due to driver unloading and loading. They observed a loading delay of over two seconds at the popular *e1000* driver[1] shipped with Linux[2]. To reduce the downtime, Zhai et al. [83], who first introduced hot plugging by a virtual ACPI hotplug implementation, propose a concept that uses Linux's bonding driver in the guest. This driver is originally intended to aggregate multiple interfaces into a single logical network interface configurable with hot standby or load balancing [8]. Zhai et al. utilized this driver to enslave a para-virtualized interface as a hot standby device and the pass-through device as the primary interface into a single network interface. This network interface has a single MAC

[1]For a wide range of *Intel Ethernet* adapters and *Intel* LOMs (LAN on Motherboard)

[2]The authors used a Linux kernel in version *2.6.18-xen* in their tests

address. With this setup, the para-virtualized device becomes active whenever the primary pass-through device is temporary unplugged due to a live migration. An analogical approach is also proposed by a solution brief of *Solarflare* [74]: They provide a driver implementation of their SR-IOV capable 10 Gbit/s *Ethernet* server adapter family that introduces plugin-based failover semantic. This driver is able, similar to the Linux bonding driver, to register a para-virtualized network device as fallback and a pass-through device as primary device (and thus as accelerated device).

VMware's US Patent 8,146,082 granted by the United States Patent and Trademark Office in March 2012 [3] replaces the pass-through device with a device emulation handler in the virtual machine monitor (VMM) during a live migration. This handler simulates the device experiencing hardware errors. When the migration procedure is done, the VMM generates an error that reports the guest to reinitialize the corresponding hardware. Then the VMM intercepts this reinitialization request from the guest to remove the emulation handler and map the new pass-through device to the migrated VM. This way, the guest reinitializes the new device while reusing the already loaded driver. Kernel data structures associated with the device are kept in memory. In the end, this approach reduces VM downtime that was caused by complete driver reloading. It also allows systems running on top of the driver to remain connected to it since kernel objects are reused.

Kadav et al. [30] showed a complete different approach with similar benefits as the device emulation handler of *VMware*, the bonding, and the plug-in driver implementation: In their work they propose a passive shadow driver which mainly capture all function calls to the device driver continuously. This shadow driver becomes only active during a migration, when the VMM injects an upcall after a migration: In this mode, the driver fields requests from the kernel until the actual driver has been restarted or even a new driver for a different pass-through device on the migration target host has been started. With the captured information about the previous driver state, the shadow driver transforms the new device driver into this state. After that, it returns itself back to passive mode. This way, software using the driver is unaware that the network device has been replaced and a migration to a different (but compatible) network device driver is also possible.

To our best knowledge, the first works that utilize self-virtualized devices (see also Section 2.1.3 on page 12) for migration with PCI Express SR-IOV [59] were proposed by Pan et al. [53] and Dong et al. [11]. However, Pan et al. CompSC study is not exactly focused on SR-IOV but a prototype was implemented on it. Device reinitialization is completely done by the guest's driver on first device access after a migration. For this purpose, their approach introduces a shared memory area between guest and the VMM. Every time when a migration happens, the VMM deposits a copy of device registers into this shared area. With this information and the internal driver state, the driver is able to reinitialize the new hardware. A counter variable, that is incremented by the VMM after a migration, indicate the guest driver that the underlying hardware was exchanged. Their work showed that in case of SR-IOV the master driver gets involved in slave device migration since a subset of the slave device state is held and managed in the master driver (e.g., MAC address).

Dong et al. went a complete different with their *ReNIC* idea [11]: Their work proposes an architectural extension for SR-IOV capable *Ethernet* devices to support state migration in the hardware. For this purpose, the master device is extended with an additional interface to extract and restore the hardware state of a slave device. During a migration, the involved slave devices are transformed into a newly introduced *clone mode* which is similar to our *device suspension* that ensures state stability (compare with Section 3.3.1 on page 36). Then, the VMMs utilize the hardware interfaces at the master driver to transport the hardware state data during a migration. Additionally, Dong et al. introduced a dirty bit for IOMMU entries so that the VMMs keep track of memory pages that were dirtied by device access, such as DMA. With this architectural modification, *pre-copy migration* is applicable even with device pass-through (compare with Section 2.2.2 on page 17). In the end, the idea of *ReNIC* enables, like our approach, guest-transparent migration of self-virtualized devices.

5.2 InfiniBand-based Approaches

Migration of directly assigned *InfiniBand* adapters has been studied more rarely compared to migration of *Ethernet* hardware. In contrast to *Ethernet* and shown in Section 2.3 on page 20, the *InfiniBand* architecture assumes central network management, stable location dependencies, and channel adapters (CAs) utilizing protocol offloading. For instance, *InfiniBand* routers and switches are configured by a so called Subnet Manager (SM), a management entity in each *InfiniBand* fabric. Moreover, address assignment is closely related to the hardware adapter so that addresses cannot be simply migrated with the VM instance. Further, *InfiniBand*'s protocol offloading introduces additional challenges, because some changes to the local hardware states may also involve changes on remote parties (e.g., destroying a queue pair (QP)). Recent *InfiniBand* migration approaches circumvent the actual problem of hardware state migration by adding reinitialization routines at certain points of the guest's *InfiniBand* stack.

The earliest work of Huang et al. [23] concerning *InfiniBand* host channel adapter (HCA) migration is based on an accelerated para-virtualized HCA implementation from Liu et al. [40]. This implementation can roughly been seen as a software realization of the shared port model introduced in Section 2.3.5 on page 25. It extends OS-bypass capabilities of the hardware to get a VMM-bypass semantic. Huang et al. [23] introduce three parts to it to support migration: (1) modifications to the user level communication libraries which allow to suspend and resume communication, (2) modifications to the guest device drivers that free and reallocate communication resources before and after a migration, and (3) a central instance for coordinating a migration and keeping track of VM locations. The proposed concept circumvents problems arising from namespaces that are bound to the local HCA (such as queue pair numbers (QPNs), LIDs, and memory keys) with virtualization: Whenever an application performs a request that returns or requires a QPN, memory key, or a remote LID of communication partners, a virtual QPN, memory key, or LID is used instead of the real one. Especially the virtual LID is unique within a cluster. It can also be seen as an identifier of a VM. Whenever a VM migrates, the coordination instance triggers all endnotes that are connected to the according VM to suspend the communication. Then, the VM that will be migrated frees the HCA resources and the VMM migrates the VM to another VMM host. After this is done, the migrated VM resetups all dependent resources and reestablishes all previous connections. At last, the coordination instance triggers all previously suspended VMs to resume communication. This virtualization approach of resources has the advantage that applications running on top of the communication libraries can still use the same identifiers to address equivalent resources after a migration. Thus, this migration approach is transparent at the application layer. However, a drawback is that communication between endnodes are only reasonable when each endnode supports the introduced extensions.

Scarpazza et al. [71] used a different way. In their work, they propose an approach where HPC applications are aware of migration and are responsible to reestablish their communication setup. Because of this, no translation mechanism is needed. In their virtual cluster checkpointing approach, the complete cluster is suspended before VM migrations are performed. However this proceeding is carried up to the application level. This way, also the applications running on the cluster have to reach a special checkpoint. This checkpoint ensures that all communication was drained over the *InfiniBand* interconnect and each HPC application has reached a quiescent state before the depnding VM gets suspended. After every VM of the virtual cluster was finally suspended, the VM migrations are performed. Each VM returns to normal execution afterwards. The still halted HPC applications are triggered by a resume signal, so that they reestablish their communication setup and hit a synchronization barrier afterwards. Only after all HPC applications reached this barrier which means that the communication setup is finally reestablished the applications return to normal execution. Especially, this prototype proposes a cluster-global view to perform a migration. However, applications need to be implemented with explicit checkpoints.

To our best knowledge, the first work that utilized self-virtualized *InfiniBand* adapters, is Guay's et al. [17] live migration prototype with SR-IOV capable *InfiniBand* HCAs. Their work was presented on a *OpenFabrics Alliance* workshop in 2012. It primarily analyzes *Mellanox's* SR-IOV capable *InfiniBand* HCAs implementing the shared port model for slave device migration towards the hotplug approach of *Ethernet*-based works. Since this is not directly realizable, they proposed two ways to introduce workarounds: (1) in a bottom-up way and (2) a top-down way. In the bottom-up way, first workarounds are placed close to the hardware and subsequently further workarounds are placed in higher levels of the *InfiniBand* software stack. In the top-down way, this is performed in reverse order. In both ways, the authors used a (almost) local three-stage strategy to perform the migration: (1) the slave device is detached at first, then (2) the according VM is migrated and finally (3) the new slave device is attached. In the bottom-up way, Guay et al. proposed changes up to the user level library: With it, the hardware QP contexts are released before slave device unplugging and reallocated after slave device plugging. Further, unplugging is performed when all send operations were completed and the QPs are in a quiescent state. QPN and memory keys are virtualized at user-library layer, so that applications can continue using their device handles afterwards. Because connected remote QPs are normally transit in an error state when the peer QP is just destroyed, the VM that is migrated generates an event to suspend the remote QP beforehand. Finally, newly created QPs on the migration target reestablish the connections. In the top-down way, Guay et al. assumed that upper-layer protocols are responsible to provide fail-over mechanisms since the *InfiniBand* software interfaces are seen as a black box. This way, they could only utilize datagram transport services which is also less generic than the bottom-up approach.

Similar to Kadav et al. [30] measurements, Guay's et al. results showed a comparable service downtime of 2.7 seconds due to driver unloading and loading. In the authors opinion, the virtual switch model seems to be a better architecture where performance optimizations are more feasible because, for instance, namespaces are isolated per slave HCA.

To our best knowledge, there is no previous approach proposing *InfiniBand* HCA migration in a guest-transparent way to get the full value of virtualization. Guay's et al. analysis was based on a local migration strategy assumption and was introducing their modification starting at the user-library level. In contrast to that, we solved transparent live migration in a software way by utilizing hardware self-virtualization at the VMM layer. Our design also benefits from the shared port model of *InfiniBand* HCAs since the master driver has - per design - control over the assigned slave devices (review Section 3.5 on page 40).

6 Conclusion

High performance computing infrastructure as a service (HPC IaaS) provides on-demand scalability and flexibility, also referred to as elasticity, to the area of high performance computing (HPC). It is an upcoming trend in the computer service industry that promises new possibilities and opportunities for users and providers.

Since recent research identified I/O network performance as the main potential performance bottleneck [19, 33] of running HPC workloads on traditional *Ethernet*-based clouds, former research [21, 24] suggest to implement HPC IaaS with a high performance interconnect, such as *InfiniBand*, and to thereby keep interconnect efficiency with low virtualization overhead. Such a hardware provide cutting-edge communication performance by utilizing aggressive protocol offloading, OS-bypass technologies, and advanced features such as remote DMA. As we discussed in this work, they substantially complicate the task of transparently migrate a virtual machine to another host. However, to get the full value for HPC virtualization, a solution for transparent live migration of HPC virtual machines (VMs) using an HPC interconnect hardware is required.

We address this challenge by proposing a novel design for device state live migration that utilizes recent interconnect adapter generations which implement hardware self-virtualization with unequal device exposure (see Section 2.1.3 on page 12). This feature solves resource sharing in hardware while a dedicated device interface, called master device, is responsible for the management of the rest of exposed interfaces, called slave devices.

Our approach consists of two orthogonal components: (1) a migration strategy and (2) a novel interface at the master device driver that we call migration assistance interface (MAI).

We analyzed that for a transparent migration, the strategy may involve remote parties in the interconnect adaption process depending on the actual migrated hardware. In such a case, we consider the strategy as global. In contrast to this, a local strategy performs necessary interconnect adaption completely transparent to interacting remote parties. Since we assume that in HPC IaaS each endnode of a virtual HPC cluster is virtualized, we proposed a global migration strategy that suspends this cluster during migration and involves the remote virtual machine monitors (VMMs) in the migration process.

MAI is thereto utilized by the VMMs for hardware state extraction and restoration of the assigned slave device whose state is migrated. This way, the actual extraction and restoration is done in the master driver so that device knowledge still remains in the particular device drivers.

Since this interface is the key component of our approach, we evaluated the feasibility of integrating MAI in a recent VMM, KVM/QEMU. We showed that MAI integrates well in QEMU by demonstrating it in a test scenario. This way, such an interface extends a VMM indirectly with device knowledge.

6.1 Future Work

Since we know that a software state migration interface, such as MAI, integrates well in a VMM design like KVM/QEMU, we will continue our research in the actual implementation of state extraction and restoration. Specifically, the feasibility of our proposed *InfiniBand* specific slave device state migration approach will answer our question if self-virtualization, such as SR-IOV, can be utilized for guest-transparent slave device state migration in HPC IaaS. Based on our observation, we are confident that the *Mellanox ConnectX-3 VPI InfiniBand* HCA fits perfectly in our migration model. We aim at a complete prototype, based on a existing cloud management framework such as *OpenNebula* [50] to demonstrate our approach in future work.

Another open question that we will follow is if in general our approach can be utilized for guest-transparent hardware state migrations of other self-virtualized devices. For this purpose, we will analyze further device types and device implementations to get an overview and a common statement.

Glossary and Abbreviations

API
Application Programming Interface

BAR
Baseline Address Register in the PCI and PCI Express standard [55, 57]

CA
Channel Adapter in the *InfiniBand* architecture [25]

CQ
Completion Queue in the *InfiniBand* architecture [25]

CQE
Completion Queue Element in the *InfiniBand* architecture [25]

DMA
Direct Memory Access; A hardware feature of modern hardware devices to directly access physical memory of a host.

FIFO
First In First Out; A queueing principle where the elements are in the same order are are picked-up from the queue as they arrived.

GID
Global ID in the *InfiniBand* architecture [25]

GUID
Globally unique ID in the *InfiniBand* architecture [25]

HCA
Host CA, Host Channel Adapter in the *InfiniBand* architecture [25]; *InfiniBand* Channel Adapter in a processor node [25]

HPC
High Perfomance Computing

HPC IaaS
High Perfomance Computing Infrastructure as a Service, a special type of IaaS for HPC

IaaS
Infrastructure as a Service

IOMMU
MMU for I/O devices that translates

IOV
I/O Virtualization

IP
Internet Protocol; IP is a unreliable connectionless packet delivery protocol and hides a underlying physical network (e.g. *Ethernet*) by creating a virtual network view [54]

KVM
Kernel-based Virtual Machine; Linux kernel interface providing machine virtualization capabilities to user space [67]

LAN
Local Area Network

LID
Local ID in the *InfiniBand* architecture [25]

LKey
Local Key for local memory access in the *InfiniBand* architecture [25]

MAC
Media Access Control provides addressing and access control mechanisms that make it possible for several network nodes to communicate within a multiple access network on a shared medium [54]

MAI
Migration Assistance Interface, introduced by our approach to assist the VMM to extract and restore the hardware device state of a slave device of a self-virtualized device (see Chapter 3 on page 27)

MMU
A Memory Management Unit is a hardware unit that primarily translates virtual addresses into host physical addresses to provide virtual address spaces for applications

MSI
Message Signaled Interrupts in the PCI and PCI Express standard [56]

MSI-X
Extended Message Signaled Interrupts in the PCI and PCI Express standard [56]

NUMA
Non-Uniform Memory Access, a multi-processor computer architecture where a com-

mon memory address space is assembled from processor-local memories. Thus, memory access performance depends on memory location related to the performing processor [20]

OFED
OpenFabrics Enterprise Distribution, an open-source software stack that provides RDMA and kernel-bypass communication interfaces to applications [49]

OS
Operating System

PaaS
Platform as a Service

PCI
Peripheral Component Interconnect [57]

PCI Express
Peripheral Component Interconnect Express [55]

PF
Physical function, also called Master Device [59]

PKey
Partition Key in the *InfiniBand* architecture [25]

QEMU
Quick EMUlator; a system emulation software [4]. In this document we term the *QEMU-KVM* fork as QEMU [34]

QP
Queue Pair in the *InfiniBand* architecture [25]; A Queue Pair consist of a Send Work Queue and a Receive Work Queue

QPN
Queue Pair Number in the *InfiniBand* architecture [25]; Numeric identifier of a QP

RDMA
Remote DMA; DMA that is initiated by a remote network instance

RKey
Remote Key for RDMA memory access in the *InfiniBand* architecture [25]

SaaS
Software as a Service

SAN
System Area Network

SM
Subnet Manager in the *InfiniBand* architecture [25]

SMA
Subnet Manager Agent in the *InfiniBand* architecture [25]

SR-IOV
Single Root IOV [59]

TCA
Target CA, Target Channel Adapter in the *InfiniBand* architecture [25]; *InfiniBand* Channel Adapter in an I/O node [25]

TCP
Transmission Control Protocol; An upper layer protocol on top of IP. It provides a connection-oriented communication to upper layer applications including error recovery, flow control, and reliability [54]

TCP/IP
TCP/IP is a common short name for the Internet protocol stack. It names the two most important protocols: TCP and IP [54]

UAR
User Access Region; A device register region of the *Mellanox ConnectX-3 VPI* HCA which is typically mapped to user space [41]

UDP
User Datagramm Protocol; A simple upper layer protocol on top of IP which is not connection oriented, provides no flow control or even error recovery [54]

ULP
Upper layer protocol in the *InfiniBand* architecture [25]

VF
Virtual function, also called Slave Device [59]

VM
Virtual Machine

VMDq
Virtual Machine Device Queues; A hardware virtualization technique for *Ethernet* network cards proposed by *Intel* [6]

VMM
Virtual Machine Monitor, also called hypervisor. We term the complete software stack that is required to provide virtual machine abstraction as VMM

WQ
Work Queue in the *InfiniBand* architecture [25]

WQE
Work Queue Element in the *InfiniBand* architecture [25]

XaaS
Everything as a Service

Bibliography

[1] Advanced Micro Devices. *AMD I/O Virtualization Technology (IOMMU) Specification*. 2009.

[2] Advanced Micro Devices. *AMD64 Technology AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Vol. 2. 2012.

[3] AM Belay. "Migrating Virtual Machines Configured With Pass-Through Devices." In: *US Patent App. 12/410,695* (2009).

[4] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *2005 USENIX Annual Technical Conference* (2005).

[5] *CentOS*. URL: http://www.centos.org.

[6] S Chinni and R Hiremane. "Virtual Machine Device Queues." In: *Intel Corp. White Paper* (2007).

[7] Christopher Clark et al. "Live Migration of Virtual Machines." In: *NSDI '05: 2nd Symposium on Networked Systems Design & Implementation* 2.Vmm (2005). ISSN: 00371963.

[8] Thomas Davis et al. *Linux Ethernet Bonding Driver HOWTO*. 2011. URL: http://www.kernel.org/doc/Documentation/networking/bonding.txt.

[9] VD Deshmukh. "InfiniBand: A New Era in Networking." In: *IJCA Proceedings on National Conference on Innovative Paradigms in Engineering & Technology* (2012).

[10] Yaozu Dong et al. "High Performance Network Virtualization with SR-IOV." In: *Journal of Parallel and Distributed Computing* 72.11 (2012).

[11] Yaozu Dong et al. "ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication." In: *ACM Transactions on Architecture and Code Optimization* 8.4 (2012). ISSN: 15443566.

[12] *Eucalyptus*. URL: http://www.eucalyptus.com/.

[13] Armando Fox and Rean Griffith. "Above the Clouds: A Berkeley View of Cloud Computing." In: *EECS Department, University of California, Berkeley* (2009).

[14] GC Foxy, KA Hawick, and AB White. “Characteristics of HPC Scientific and Engineering Applications.” In: *Second Pasadena Workshop on System Software on Tools for High Performance Computing Environments* (1996).

[15] Abel Gordon et al. “ELI : Bare-Metal Performance for I/O Virtualization.” In: *ASPLOS ’12: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (2012).

[16] Wei Lin Guay et al. “Host Side Dynamic Reconfiguration with InfiniBand.” In: *2010 IEEE International Conference on Cluster Computing* (2010).

[17] Richard Guay, Wei Lin and Johnsen, Bjørn Dag and Torudbakken, Ola and Yen, Chien-Hua and Reinemo, Sven-Arne, Frank. “Prototyping Live Migration With SR-IOV Supported InfiniBand.” In: *OFA International Workshop 2012* (2012).

[18] P. H. Gum. “System/370 Extended Architecture: Facilities for Virtual Machines.” In: *IBM Journal of Research and Development* 27.6 (1983). ISSN: 0018-8646.

[19] Abhishek Gupta and Dejan Milojicic. “Evaluation of HPC Applications on Cloud.” In: *2011 Sixth Open Cirrus Summit* (2011).

[20] JL Hennessy and DA Patterson. *Computer architecture: A Quantitative Approach*. Fourth Edi. Morgan Kaufmann Publishers, 2011. ISBN: 9780123704900.

[21] Marius Hillenbrand et al. “Virtual InfiniBand clusters for HPC Clouds.” In: *CloudCP ’12: 2nd International Workshop on Cloud Computing Platforms* (2012).

[22] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. “Post-copy Live Migration of Virtual Machines.” In: *ACM SIGOPS Operating Systems Review* 43.3 (2009). ISSN: 01635980.

[23] W Huang, J Liu, and M Koop. “Nomad: Migrating OS-bypass Networks in Virtual Machines.” In: *VEE ’07: Conference on Virtual Execution Environments* (2007).

[24] Wei Huang et al. “A Case for High Performance Computing with Virtual Machines.” In: *ICS ’06: Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006. ISBN: 1595932828.

[25] InfiniBand Trade Association. “InfiniBand Architecture Specification Volume 1: Release 1.2.1 (Final).” In: 1.November (2007).

[26] Intel Corporation. “Intel Virtualization Technology for Directed I/O - Archticture Specification.” In: *Intel technology journal* February (2006).

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*. August. 2012.

[28] Intel Corporation. “PCI-SIG SR-IOV Primer.” In: *Intel LAN Access Division:* (2011).

[29] LE Jonsson and WR Magro. “Comparative Performance of InfiniBand Architecture and Gigabit Ethernet Interconnects on Intel® Itanium® 2 Microarchitecture-based Clusters.” In: *4th European LS-DYNA Users Conference* (2003).

[30] Asim Kadav and Michael M. Swift. “Live migration of direct-access devices.” In: *ACM SIGOPS Operating Systems Review* 43.3 (2009). ISSN: 01635980.

[31] *Kernel-based Virtual Machine (KVM)*. URL: http://www.linux-kvm.org.

[32] Avi Kivity, Y Kamay, and D Laor. “KVM: The Linux Virtual Machine Monitor.” In: *Proceedings of the Linux Symposium* (2007).

[33] Brian Kocoloski, J Ouyang, and John Lange. “A Case for Dual Stack Virtualization: Consolidating HPC and Commodity Applications in the Cloud.” In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012. ISBN: 9781450317610.

[34] *KVM - QEMU*. URL: http://wiki.qemu.org/KVM.

[35] Joshua Levasseur et al. “Standardized but Flexible I/O for Self-Virtualizing Devices.” In: ().

[36] Guangdeng Liao et al. “Software Techniques to Improve Virtualized I/O Performance on Multi-Core Systems.” In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems - ANCS ’08* (2008).

[37] *libvirt virtualization API*. URL: http://libvirt.org.

[38] *Linux RDMA and InfiniBand Development*. URL: http://www.spinics.net/lists/linux-rdma/.

[39] Liran Liss and Mellanox Technologies. “Infiniband and RoCEE Virtualization with SR-IOV.” In: *OFA International Workshop 2010* (2010).

[40] Jiuxing Liu, Wei Huang, and Bulent Abali. “High performance VMM-bypass I/O in virtual machines.” In: *ATEC ’06 Proceedings of the annual conference on USENIX ’06 Annual Technical Conference* Vmm (2006).

[41] Mellanox Technologies. *ConnectX-3 prereleased Linux driver (SRIOV-ALPHA-3.3.0-2.0.0008)*. 2012.

[42] Mellanox Technologies. “Mellanox ConnectX InfiniBand Adapter Brochure.” In: (2013). URL: http://www.mellanox.com/related-docs/products/IB_Adapter_card_brochure_c_2_3.pdf.

[43] Mellanox Technologies. *Mellanox OFED Stack for Linux - User’s Manual*. Mellanox Technologies, 2008.

[44] Mark F Mergen et al. “Virtualization for High-Performance Computing Categories and Subject Descriptors.” In: ().

[45] JC Mogul and KK Ramakrishnan. “Eliminating Receive Livelock in an Interrupt-driven kernel.” In: *Proceedings of the USENIX 1996 Annual Technical Conference* January (1996).

[46] Jack Morgenstein. *Patch for linux-rdma project that adds SR-IOV support for Infini-Band interfaces*. 2012. URL: http://www.mail-archive.com/linux-rdma@vger.kernel.org/msg11956.html.

[47] Myricom Inc. *Myrinet*. URL: http://www.myri.com.

[48] Michael Nelson, BH Lim, and Greg Hutchins. “Fast Transparent Migration for Virtual Machines.” In: *Proceedings of USENIX ’05: General Track* (2005).

[49] OpenFabrics Alliance. *OFED Overview*. URL: https://www.openfabrics.org/resources/ofed-for-linux-ofed-for-windows/ofed-overview.html.

[50] *OpenNebula*. URL: http://www.opennebula.org/.

[51] *OpenSM and InfiniBand diagnostic utilities*. URL: http://www.openfabrics.org/downloads/management/.

[52] *OpenStack*. URL: http://www.openstack.org/.

[53] Zhenhao Pan and Y Dong. “CompSC: Live Migration with Pass-through Devices.” In: *VEE ’12: Conference on Virtual Execution Environments* (2012).

[54] Lydia Parziale et al. *TCP/IP Tutorial and Technical Overview*. 8th. IBM Corporation, 2006.

[55] PCI-SIG. *PCI Express Base Specification Revision 3.0*. 2010.

[56] PCI-SIG. *PCI Local Bus pecification Revision 2.3 MSI-X ECN*. 2003.

[57] PCI-SIG. *PCI Local Bus Specification Revision 3.0*. 2002.

[58] *PCI SIG*. URL: http://www.pci-sig.com.

[59] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification Revision 1.1*. 2010.

[60] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. “The Case of the Missing Super-computer Performance.” In: *SC ’03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (2002).

[61] Gerald J. Popek and Robert P. Goldberg. “Formal requirements for Virtualizable Third Generation Architectures.” In: *Communications of the ACM* 17.7 (1974). ISSN: 00010782.

[62] Tim Pritlove and André Przywara. *CRE092 Virtualisierung*. 2008. URL: http://cre.fm/cre092.

[63] Tim Pritlove and Tobias Rodäbel. *CRE176 Cloud Computing*. 2011. URL: http://cre.fm/cre176.

[64] André Przywara. "Live und in Farbe: Live Migration." In: *Chemnitzer Linux-Tage 2010* (2010).

[65] *ps(1) - Linux man page*. URL: http://linux.die.net/man/1/ps.

[66] *Quick EMUlator (QEMU)*. URL: http://wiki.qemu.org.

[67] Qumranet. "KVM : Kernel-based Virtualization Driver." In: *White Paper* (2006).

[68] Himanshu Raj and Karsten Schwan. "Implementing a Scalable Self-Virtualizing Network Interface on an Embedded Multicore Platform." In: *Proceedings of WIOSCA 2005* (2005).

[69] RedHat. *RedHat Enterprise Linux (RHEL)*. URL: http://www.redhat.com/products/enterprise-linux.

[70] JR Santos and Yoshio Turner. "Bridging the Gap between Software and Hardware Techniques for I/O Virtualization." In: *Proceedings of the USENIX Annual Technical Conference* (2008).

[71] DP Scarpazza and P Mullaney. "Transparent System-level Migration of PGAS Applications using Xen on InfiniBand." In: *IEEE International Conference on Cluster Computing* (2007).

[72] Amit Shah et al. "PCI Device Passthrough for KVM." In: *KVM Forum 2008* (2008).

[73] Tom Shanley. *InfiniBand Network Architecture*. Addison-Wesley, 2007.

[74] Solarflare Communications. "Solarflare 10G Ethernet Server Adapters Deliver Unified Single-Root I/O Virtualization (SR-IOV) for Redhat Linux KVM." In: *Solution Brief* ().

[75] Andreas Ströbel. "XaaS – Everything as a Service?" In: *bt magazin* (2011).

[76] J Sugerman, G Venkitachalam, and BH Lim. "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor." In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (2001).

[77] *The Linux Kernel Archives*. URL: http://www.kernel.org.

[78] *The Message Passing Interface (MPI) standard*. URL: http://www.mcs.anl.gov/research/projects/mpi/.

[79] *TOP500 Supercomputer List: November 2012*. URL: http://www.top500.org/lists/2012/11/.

[80] Paul Willmann, Jeffrey Shafer, and David Carr. “Concurrent Direct Network Access for Virtual Machine Monitors.” In: *HPCA ’07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (2007).

[81] *Xen PCI Passthrough*. URL: `http://wiki.xen.org/wiki/Xen\_PCI\_Passthrough`.

[82] BA Yassour, M Ben-Yehuda, and Orit Wasserman. “Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines.” In: *IBM Research Report* (2008).

[83] Edwin Zhai, GD Cummings, and Yaozu Dong. “Live Migration with Pass-through Device for Linux VM.” In: *OLS’08: The 2008 Ottawa Linux Symposium* (2008).