Karlsruher Institut für Technologie

Towards Virtual InfiniBand Clusters with Network and Performance Isolation

Diplomarbeit
von

cand. inform. Marius Hillenbrand

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Betreuende Mitarbeiter:	Dr.-Ing. Jan Stoess
	Dipl.-Phys. Viktor Mauch

Bearbeitungszeit: 17. Dezember 2010 – 16. Juni 2011

Abstract

Today's high-performance computing clusters (HPC) are typically operated and used by a single organization. Demand is fluctuating, resulting in periods of underutilization or overload. In addition, the static OS installation on cluster nodes leaves hardly any room for customization. The concepts of cloud computing transferred to HPC clusters — that is, an Infrastructure-as-a-Service (IaaS) model for HPC computing — promises increased flexibility and cost savings. Elastic virtual clusters provide precisely that capacity that suits actual demand and workload.

Elasticity and flexibility come at a price, however: Virtualization overhead, jitter, and additional OS background activity can severely reduce parallel application performance. In addition, HPC workloads typically require distinct cluster interconnects, such as InfiniBand, because of the features they provide, mainly low latency. General-purpose clouds with virtualized Ethernet fail to fulfill these requirements.

In this work, we present a novel architecture for HPC clouds. Our architecture comprises the facets node virtualization, network virtualization, and cloud management. We raise the question, whether a commodity hypervisor (the kernel-based virtual machine, KVM, on Linux) can be transformed to provide *virtual cluster nodes* — that is, virtual machines (VMs) intended for HPC workloads. We provide a concept for cluster network virtualization, using the example of InfiniBand, that provides each user with the impression of using a dedicated network. A user can apply a custom routing scheme and employ recursive isolation in his share of the network. However, he remains constraint to his virtual cluster and cannot impair other users — we verify this claim with experiments with an actual InfiniBand network. We discuss the new challenges that cluster networks bring up for cloud management, and describe how we introduce network topology to cloud management. A prototype for automatic network isolation provides a proof of concept.

Deutsche Zusammenfassung

Computercluster für das Hochleistungsrechnen werden heute üblicherweise von einzelnen Organisationen betrieben. Die Rechenkapazität solcher Systeme ist festgelegt, weswegen schwankende Arbeitslasten abwechselnd zu zeitweiliger Überlast oder zu Phasen von Leerlauf führen. Betriebssysteme und Ablaufumgebungen sind fest auf den Knoten eines Computerclusters installiert und lassen nur wenig Spielraum für anwendungs- oder nutzerspezifische Anpassungen. Überträgt man jedoch die Konzepte von *Cloud Computing* – die dynamische und automatisierte Bereitstellung von Ressourcen sowie die Abrechnung auf Nutzungsbasis – auf Computercluster, so verspricht das eine deutlich erhöhte Flexibilität und eine Verringerung von Kosten. Virtuelle Computercluster würden so zu jedem Zeitpunkt genau die Rechenkapazität bereitstellen, die benötigt wird.

Diese Flexibilität und Variationsmöglichkeit der Rechenkapazität (die sog. *Elastizität*), hat jedoch ihren Preis: Mehraufwand, variierende Ausführungszeiten einzelner Iterationen und mehr Hintergrundaktivität in Betriebssystem und Virtualisierungsschicht können die Leistung paralleler Anwendungen deutlich herabsetzen. Darüber hinaus benötigen viele Anwendungen für das Hochleistungsrechnen speziell für Computercluster entworfene Netzwerktechnologien, weil diese sehr geringe Latenzzeiten erreichen und erweiterte Kommunikationsprimitive bieten. Bestehende Lösungen für *Cloud Computing* können diese Anforderung nicht erfüllen, weil sie nur virtualisierte Ethernet-Netzwerke anbieten.

Die vorliegende Arbeit beschreibt eine neuartige Architektur für *Cloud Computing* für das Hochleistungsrechnen, bestehend aus drei Teilaspekten: Virtualisierung der Cluster-Knoten, Virtualisierung des Cluster-Netzwerks und Management der resultierenden *Cloud.* Die Arbeit untersucht, ob eine gängige Virtualisierungsschicht (KVM unter Linux) für die Bereitstellung virtueller Knoten für Computercluster herangezogen werden kann. Weiterhin wird ein Konzept für die Virtualisierung der Cluster-Netzwerktechnologie InfiniBand vorgestellt, das einem Benutzer den Eindruck vermittelt, er hätte weitreichenden und exklusiven Zugriff auf ein physisches Netzwerk. Tatsächlich nutzt er jedoch nur einen Teil des physisch vorhandenen Netzwerks und kann andere Nutzer nicht störend beeinflussen. Die Arbeit bespricht zudem die Herausforderungen, die eine automatisierte Verwaltung eines Cluster-Netzwerks mit sich bringt, und beschreibt geeignete Erweiterungen bestehender Verwaltungssysteme für *Cloud Computing.*

Teil dieser Arbeit ist die Evaluation des Einflußes unterschiedlicher Konfigurations-Optionen des Linux-Kernels auf die Hintergrundaktivität des Systems *(OS noise)* und die Leistung von Anwendungen. Zudem wird an einem realen InfiniBand-Netzwerk untersucht, ob sich eine Netzwerk-Isolation tatsächlich praktisch erzwingen lässt und inwieweit bestehende Werkzeuge hierfür geeignet sind. Die automatisierte Verwaltung eines Cluster-Netzwerks, als integraler Bestandteil eines Verwaltungssystems für *Cloud Computing,* wird an Hand eines Prototypen demonstriert, der in einem InfiniBand-Netzwerk die Netzwerk-Isolation zwischen virtuellen Computerclustern konfiguriert.

Contents

1 Introduction

Today's high-performance computing clusters (HPC) are typically operated and used by a single organization. Demand is fluctuating, resulting in periods where physical resources are underutilized or overloaded. In addition, the static OS installation on cluster nodes leaves little room for customizations of the runtime environment for different jobs. The concepts of cloud computing transferred to HPC clusters — that is, an Infrastructure-as-a-Service (IaaS) model for HPC computing — promises increased flexibility and cost savings. It enables the progress away from physically owned but underutilized HPC clusters designed for peak workloads to virtualized and elastic HPC resources leased from a consolidated large HPC computing center working near full capacity. Elastic virtual clusters provide compute capacity scaled dynamically to suit actual demand and workload. At the same time, the pay-as-you-go principle of cloud computing avoids the huge initial investments that are inevitable with physically owned clusters, and causes costs only for actual use of computing capacity. Virtualization allows to dynamically deploy a fully custom runtime environment for each user, customized from the OS kernel up to libraries and tools.

In practice, however, providing cloud-based HPC raises difficult challenges. Virtualization, the core technique of contemporary general-purpose IaaS offerings, has a high associated overhead and, even more important, may lead to unpredictable variations in performance [63, 93]. Although such overhead and jitter may be tolerable for standard workloads, running HPC tasks used to predictable performance delivery and microsecond-latencies in a compute cloud becomes a non-trivial problem: it is well known that jitter and OS background activity can severely reduce parallel application performance [27, 67, 89]. Therefore, a compute cloud for HPC workload must strive to incur minimal OS background activity.

Furthermore, contemporary compute clouds implement network isolation and bandwidth sharing with general-purpose Ethernet overlays. This approach achieves elasticity of network resources for standard server workloads. Compute clusters often employ distinct cluster interconnect networks, however, because of their performance characteristics and advanced features (such as remote DMA transfers), which the virtualized Ethernet of general-purpose clouds fails to provide. So, an IaaS model for HPC faces a new challenge: it must incorporate the management of cluster interconnects to provide the low latency and feature set that HPC users expect and require for their applications. It must provide virtual machines (VMs) with access to cluster interconnect adapters, which is not yet a commodity, like Ethernet virtualization. In addition, multi-tenancy in HPC clouds requires network isolation, and network resources must be shared in a way that fulfills the quality of service (QoS) requirements of HPC applications.

In this work, we present a novel architecture for HPC clouds that provide virtual and elastic HPC clusters. Our architecture comprises the three facets node virtualization, network virtualization, and HPC cloud management. We raise the question, whether a commodity hypervisor,

the kernel-based virtual machine (KVM) on Linux, can be transformed to support our architecture. We explore the state of the art in virtualized InfiniBand access and discuss how we customize a Linux host OS to provide *virtual cluster nodes* — that is, VMs intended for HPC workloads that altogether form a virtual compute cluster. In our evaluation, we examine the *OS noise* (OS background activity) in different Linux kernel configurations and assess how OS noise and kernel configuration affect application performance and overhead.

We provide an extensive concept for InfiniBand network virtualization that provides each user with the impression of using a dedicated physical network with full access to all configuration features. A user can apply a custom packet routing scheme, employ recursive isolation in his share of the network, and assign custom node addresses. His configuration settings affect only his share of the network, however, and cannot impair other users. We evaluate how effective network and performance isolation can be enforced in a real InfiniBand network. For this purpose, we examine the practically available protection features that help to restrain a user from altering the network configuration outside his virtual cluster.

With regard to the third aspect of our architecture — that is, HPC cloud management — we discuss the additional challenges that the performance characteristics and peculiarities of cluster networks bring up, using the example of InfiniBand. To face these challenges, we extend existing cloud management frameworks with information about the cluster network topology. We incorporate this information into cloud management, so that we can consider network distance (to minimize it) and the restrictions of InfiniBand QoS mechanisms (not more than 15 virtual clusters share a network link) in VM placement. As a proof of concept, we present a prototype that combines cloud and cluster network management and automatically configures InfiniBand network isolation based on the grouping of VMs to virtual clusters.

Compute clusters[1] clearly are the prevalent architecture in today's HPC computing. Our focus on distinct cluster interconnects, such as InfiniBand, meets the state of the art in building high-end clusters. The TOP500 supercomputer list[2] currently lists 414 cluster systems among the 500 most powerful supercomputer systems — that is, 83%. Gigabit Ethernet and InfiniBand are the most commonly used network technologies for clusters. The 16 highest ranked clusters all employ InfiniBand, as do 77% of the top 100 cluster systems.

Employing virtualization in a HPC IaaS model introduces a high level of flexibility and provides for elasticity. These achievements are paid with a reduction of performance caused by virtualization overhead, however. Considering this trade-off, the question arises, whether one cannot implement isolation and elasticity for a multi-tenant HPC cloud at the OS level, and thus avoid the overhead of a distinct virtualization layer. There are several concepts that provide isolated environments in a single OS, such as OpenVZ, which is known to provide less overhead than virtualization in some applications [17]. Our approach for InfiniBand network virtualization is applicable at the OS level, too.

However, besides incurring overhead, virtualization provides distinct advantages for HPC, which have been pointed out in former work already [28, 37, 59]. Virtualization allows to cus-

[1] Compute clusters, also called cluster computers, are distributed and networked multi-computer systems, see (p.546) [88].

[2] The TOP500 supercomputer list is a half-annual ranking of supercomputers based on the high-performance linpack benchmark [24, 60].

tomize the runtime environment of each job (e.g., to run legacy HPC applications in a legacy OS), it enables new transparent fault tolerance mechanisms, and it maintains isolation when regular users are granted superuser privileges in a VM. We shall elaborate the advantages that virtualization provides for HPC in greater detail in Section 2.2 on page 6. Further, compute-intensive workloads typically show acceptable virtualization overhead, as noticed in [28] already — in the course of our evaluation, we have observed virtualization overheads of typically less than 4 % (with a maximum of 10 % only in a single configuration), see Section 4.2.1 on page 60.

Following this introduction, we present background information on virtualization and the HPC application model we follow in this work, amongst others, and discuss related work on virtual HPC clusters and advanced network virtualization concepts in Chapter 2 on page 5. In Chapter 3 on page 17, we introduce our architecture for HPC clouds and discuss our architecture's individual facets node virtualization (in Section 3.2 on page 20), network virtualization (in Section 3.3 on page 30), and HPC cloud management (in Section 3.4 on page 46). We complement our approach with the discussion of our prototypic HPC cloud and the practical evaluation of some of its building blocks for their fitness for purpose in Chapter 4 on page 53. In Chapter 5 on page 85, we shall conclude our work and give a summary of our results.

2 Background and Related Work

In this section, we prepare and lay down some background and provide references to former work that we build upon. In addition, we present related work and point out how our contribution differs in Section 2.7 on page 13.

Cloud computing is a trend that we want to extend to high-performance computing (HPC). Virtualization is the basic building block of cloud computing. We introduce these highly related topics in Section 2.1 on page 5. Virtualization offers some distinct advantages for HPC, despite the performance degradation incurred by virtualization overhead. These advantages have been well discussed in former work, which we summarize in Section 2.2 on page 6. Virtual Machines (VMs) require virtual I/O devices, because they are typically prohibited access to physical I/O devices. In particular, virtualization of network devices is very critical for the performance of server workloads and has consequently encouraged much research (especially Ethernet virtualization). Since we utilize virtualized access to distinct cluster networks, a related but less thoroughly researched issue, we discuss device virtualization using the example of Ethernet virtualization in Section 2.3 on page 8. Throughout this work, we consider HPC applications as defined by the bulk-synchronous single-program-multiple-data application model, which we therefore introduce in Section 2.5 on page 10. In addition, we refer to former research on the performance characteristics of such applications. Finally, we discuss some publications that analyze contemporary general-purpose clouds for various types of HPC workload in Section 2.6 on page 12.

In this work, we employ InfiniBand as the cluster interconnect of our HPC cloud. However, we shall introduce the concepts, properties, and mechanisms of InfiniBand during the course of this work — each aspect in the context where we employ it — instead of covering them isolated in this chapter.

2.1 Cloud Computing and Virtualization

Cloud computing is a current trend in the IT industry. Users of applications and computing infrastructure move from owning and operating applications and hardware by themselves towards utilizing software and computing infrastructure that is offered as a service by cloud providers (referred to as Software-as-a-Service, SaaS, and Infrastructure-as-a-Service, IaaS). A user can order such services dynamically on demand and only has to pay for their actual use. According to [5] and [16], cloud computing is the vision of utility computing coming true.

One of the foundations of cloud computing is the virtualization of compute resources, storage, and networking (compare [5, 16]). Throughout this work, we will use the term virtual machine (VM) to refer to a virtualization of the complete instruction set architecture of the host system.

An OS running inside a VM, often termed *guest OS* has the impression of running on a dedicated computer. The software layer that implements this virtualization is called hypervisor or virtual machine monitor. See Tanenbaum's classic text book, *Modern Operating Systems, 3rd edition,* (p. 65–69) [88] and (p. 566–587) [88], for a more in-depth introduction to VMs.

There are several approaches to system level virtualization — an overview is presented by Smith and Nair in [83]. An early approach, called trap-and-emulate or *classic* virtualization, has initially been developed for the IBM mainframe architecture (compare [88]) and has been formally described by Popek and Goldberg in their classic paper [69]. A more recent approach is binary translation, which has become popular in the virtualization products from VMware [1]. It has been devised to overcome limitations of the x86 architecture that make trap-and-emulate impossible [75].

In our work, we employ the kernel-based virtual machine (KVM) hypervisor [49]. KVM uses hardware extensions that facilitate virtualization, which have recently been introduced to the x86 architecture by Intel and AMD in the form of Intel VT [90] and AMD-V [2]). However, this concept has also been used first on the IBM mainframe architecture, called interpretive execution [35]. It provides a separate machine state for the execution of a guest system's instructions. The CPU can execute some privileged instructions of the guest OS without involving the hypervisor, while constraining their effects to the VM (i.e., its dedicated machine state).

2.2 Benefits of Virtualization for High-Performance Computing

System level virtualization provides some distinct advantages for high-performance computing. In this section, we present three published proposals for the use of virtualization for HPC workload [28,37,59]. All three proposals mention several advantages of virtualization for HPC, which we summarize here.

Figueiredo, Dinda and Fortes have been first in proposing virtualized environments for HPC workload. They suggested to deploy jobs in grid computing as VMs running inside a hypervisor on compute nodes [28]. Mergen and associates propose virtualization to combine legacy OS environments and custom lightweight runtime environments for HPC workloads [59] and thereby combine the advantages of both options. Performance benefits from custom environments, and existing legacy OSs provide a rich functionality, covering wide hardware support and tool support for debugging. Huang and associates developed a batch job management system that runs each HPC job in a VM that provides the required environment for the job [37]. They further proposed VMM-bypass I/O, direct I/O access of guest applications, to reduce virtualization overhead for I/O operations.

In all three proposals [28, 37, 59], the respective authors present several advantages that virtualization offers for HPC:

- Virtualization allows to use specialized environments for jobs, ranging from libraries down to the OS kernel, both customized installations of a standard OS, such as Linux,

and specific lightweight OSs for HPC. Several specialized runtime environments can coexist at the same time in a physical cluster and even on the same physical host.

- VMs allow to provide and preserve binary-compatible runtime environments for legacy applications, without putting constraints on other applications or on the use of physical resources — in contrast to reserving some physical nodes for a legacy environment.
- Such customized runtime environments are much easier and faster to start and stop in a VM than on a physical host, because hardware re-initialization is avoided.

In contrast to server workloads, HPC applications are typically run as batch jobs, which are transient, and thus require a certain runtime environment only temporary. The HPC applications used in jobs are reoccurring however. For example, an oceanographer may run a simulation software for ocean currents repeatedly, each time with different parameters and input data. So, the required job runtime environments are reoccurring, too. Time-consuming reconfigurations of the OS installation on a system are undesirable when they are required for individual HPC jobs with a short- to medium-term lifespan, whereas they are quite tolerable for server workloads, which are expected to operate for a long duration. As an alternative, a set of VM images can be prepared to provide each HPC application with a matching runtime environment.

Further advantages of using system level VMs (compare [28, 37, 59]) affect isolation, productivity, and fault tolerance, amongst others:

- Isolation is provided by the hypervisor, in the virtualization layer below the (guest) OS. So, users of a VM can be granted superuser privileges inside the guest OS, if required, without breaking isolation between separate VMs or compromising the integrity of the host.
- VMs implicitly form a resource principal and an entity in the hypervisor's resource management that covers all the resource utilization inside the guest, including OS activity, which is traditionally hard to associate with processes [11] (e.g., handling received packets in the network protocol stack on behalf of a process).
- Virtualization can facilitate testing and debugging of parallel applications by providing many virtual nodes, as required to detect scalability problems, on much fewer physical nodes (e.g., by mapping many single-processor virtual nodes to few multi-core physical nodes). In addition, virtualization allows to debug and monitor VMs from the outside, thereby avoiding disturbance from integrating debugging code into the OS or application under test.
- Virtualization adds a layer to implement fault tolerance, partially transparent to the guest OS. A hypervisor can improve software reliability with checkpoint/restart schemes for VMs. At the same time, it can shield VMs against (some) hardware failures and thereby reduce the need to implement fault tolerance mechanisms inside a guest OS.

In summary, virtualization enables new features for HPC workload, such as providing regular users with superuser privileges inside a VM or transparent fault tolerance mechanisms in the hypervisor. Virtualization increases flexibility and thereby allows to adapt existing compute resources dynamically to more diverse workloads with differing required runtime environments.

2.3 Ethernet Device Virtualization

A hypervisor typically denies a guest OS direct access the I/O devices of the host for the purpose of isolation. So, the hypervisor has to provide the guest OS with a virtual I/O subsystem, or controlled access to the host's I/O devices. An important class of virtualized devices are Ethernet adapters that allow several VMs access to a TCP/IP network via a physical Ethernet adapter in the host, because TCP/IP networks on Ethernet are a commodity in server computing today. Consequently, much research has been done on Ethernet virtualization. Although our work focuses on the related topic of virtualizing cluster networks, we introduce device virtualization techniques using the example of Ethernet virtualization, because it allows us to present the evolution of techniques from full emulation to virtualization support in devices.

When a guest OS issues instructions for I/O access, these instructions trap to the hypervisor, because they are privileged. One approach for device virtualization emulates physical devices in the hypervisor based on these traps (called full emulation). The emulated device has the same software interface as an existing physical one, so regular device drivers can be used in the guest OS. As a disadvantage, the emulation causes significant overhead (which can be mitigated to a certain degree [87]).

Instead of emulating real hardware, a hypervisor can provide an interface specially designed for virtual devices. This approach is called paravirtualization [12, 94] and reduces overhead, compared to full emulation. However, it poses the disadvantage of requiring the development of new drivers that support the new virtual device for all potential guest OSs. Xen [12] employs a split-driver model, where a backend driver in a dedicated device driver VM has exclusive access to physical hardware, and frontend drivers in guest OSs communicates with the backend driver to perform I/O. Network packets are first transfered between guest OS and device driver VM before they are sent to the network card, which causes notable overhead with high-speed networks [18]. Several improvements have been proposed, such as avoiding copy operations [58] or fixing performance problems in the implementation [76], amongst others.

In 2007, several publications brought forward the idea to involve physical Ethernet adapters in the process of virtualization [54,72,79] , under different notions, such as self-virtualized devices [72] or concurrent direct network access [79]. The device offers several virtual interfaces for device drivers, it is aware that several OS instances access it, and it multiplexes the physical resources between the virtual interfaces, including the sorting of received packets to different receive queues.

The PCI Special Interest Group, the standardization organisation behind the Peripheral Component Interconnect (PCI), has defined a generic interface for self-virtualized devices, called Single Root I/O Virtualization (SR-IOV) [82]. A physical PCI device with enabled SR-IOV support behaves like several logical devices, each with its own device interface. The first logical device is called physical function (PF). The PF is used to enable SR-IOV and to manage the other logical devices, called virtual functions (VFs). Access to the PF is typically restricted to the hypervisor or a device driver OS. A VF represents a complete software interface to a PCI device, which can be used by a guest VM to access the device independent from other VMs or the hypervisor.

A hypervisor grants a VM transparent access to a VF using a mechanism called PCI passthrough. Basically, the hypervisor maps the memory-mapped I/O region(s) of the device into the VM's address space. PCI devices can act as bus masters and perform DMA transfers. PCI DMA transfers operate on physical addresses by default, but a guest OS does not know about physical addresses of its DMA buffers. So, with PCI passthrough alone, DMA transfers initiated by a guest OS would not work as expected. Even worse, a malicious guest OS could exploit a PCI device's DMA capability to access host physical memory and thereby break the isolation between VMs. So, additional precautions are necessary (compare [97]).

A solution is to add an address translation mechanism between PCI devices and main memory. Such an address translation between I/O devices and main memory is commonly called an I/O memory management unit (IOMMU). It uses different address mappings, based on which device performs a memory access. When a PCI device is assigned to a VM, the hypervisor configures the IOMMU to apply the VM's guest-physical to host-physical memory mapping to each DMA transfer of the respective device. As a result, the IOMMU transparently redirects and restricts each DMA transfer a guest OS orders to the memory assigned to the respective VM. This procedure applies to VFs provided by SR-IOV and to physical devices that are dedicated to a VM. IOMMUs have been implemented by Intel, called Virtualization Technology for Directed I/O [39], and AMD, called I/O Virtualization Technology [3].

Employing an IOMMU, PCI passthrough can be implemented in a device-independent way. SR-IOV and IOMMU support has been added to the hypervisors Xen [23] and KVM [22, 97]. A performance evaluation of an SR-IOV capable 10 GbE device can be found in [52] and shows increased performance compared to paravirtualization.

2.4 Cluster Network Virtualization

In the previous section, we presented the concepts employed for device virtualization using the example of Ethernet network cards. We have shown the evolution of virtualization techniques from full emulation to standardized virtualization support in devices with SR-IOV. In this section, we present former work on the virtualization of cluster networks, which has received considerably less attention than Ethernet virtualization.

Liu and associates developed and analyzed a paravirtualization approach for InfiniBand host channel adapters (HCAs) [53], termed VMM-bypass I/O. It allows a guest process to directly issue communication requests to an InfiniBand HCA, while isolation between VMs and processes is maintained. Huang has extended this approach by support for live migration of VMs that use InfiniBand in his PhD thesis [36]. This framework however spans hypervisor, guest OS, and guest MPI libraries, which all are involved in the migration. So, live migration transparent to the guest OS is not possible that way. Unfortunately, the prototypic sourcecode of this paravirtualization approach that is publicly available, has not been maintained since 2007.

When guest processes access I/O devices without involving the hypervisor, the hypervisor cannot directly enforce resource allocations for I/O utilization. Ranadive and associates proposed fair resource scheduling based on asynchronous monitoring of I/O usage and VM memory inspection to face this problem [73].

Nanos and Koziris presented paravirtualization for Myrinet cluster networks [62]. Their approach is very similar to the InfiniBand paravirtualization by Liu and associates [53], because they also used the Xen hypervisor and Myrinet is very similar to InfiniBand with regard to concepts of the software interface that has to be virtualized.

2.5 Bulk-Synchronous SPMD Model and Performance Analysis

Many applications that are typically considered as HPC workload match the bulk-synchronous single-program multiple-data (SPMD) application model [6, 45], which is the adaptation of the SPMD application model [20] to the bulk synchronous parallel machine model [92]. In this model, a parallel job consists of several processes that run concurrently and execute the same binary program. The number of processes remains static during the lifetime of a job and each process runs on a separate processor (compare [6]). Memory is distributed, as introduced by the bulk-synchronous parallel machine model, in contrast to shared-memory with the original SPMD model [20]. Therefore processes exchange data and synchronize via message-passing.

Many HPC applications can be seen simplified as cycling through two phases: Purely local computation alternates with synchronization and communication between processes, see Figure 2.1 on page 11. Synchronization primitives such as the barrier (designated in the original SPMD model [20]), are so-called collective operations that involve all processes. Every single process has to wait until all other processes have participated in the collective operation (compare [45]).

A popular, standardized framework for the bulk-synchronous SPMD model is the Message Passing Interface (MPI) [30]. It supports starting and running SPMD-style parallel jobs and defines an API for communication operations between individual processes. MPI implementations are available on a broad range of hardware platforms. One of them is Open MPI [33], which we used in our prototypic evaluation (see Section 4.1.5 on page 58). According to [33], it is based on the experiences gained from former implementations LAM/MPI, LA-MPI, FT-MPI and PACX-MPI. Its design is consistently component-based and aims to achieve high performance in distributed computing systems that are heterogeneous regarding processor architecture, network technology and protocols, run-time environment, and binary formats. Details on how Open MPI utilizes InfiniBand can be found in [81].

The performance characteristics of bulk-synchronous SPMD applications have been studied very well since its initial publication. In the context of our work, we are mainly interested in the performance implications that result from the interaction of such applications with an OS.

Jones and colleagues have analyzed the influence of OS background activity on collective synchronization operations [45]. They try to minimize this influence by overlapping background activity on all nodes. Thereby, the overall duration a parallel application is interrupted on any node (potentially causing other nodes to wait) is reduced. For this purpose, they propose OS modifications that co-schedule background activity by the OS kernel and system daemons on all nodes of a cluster.

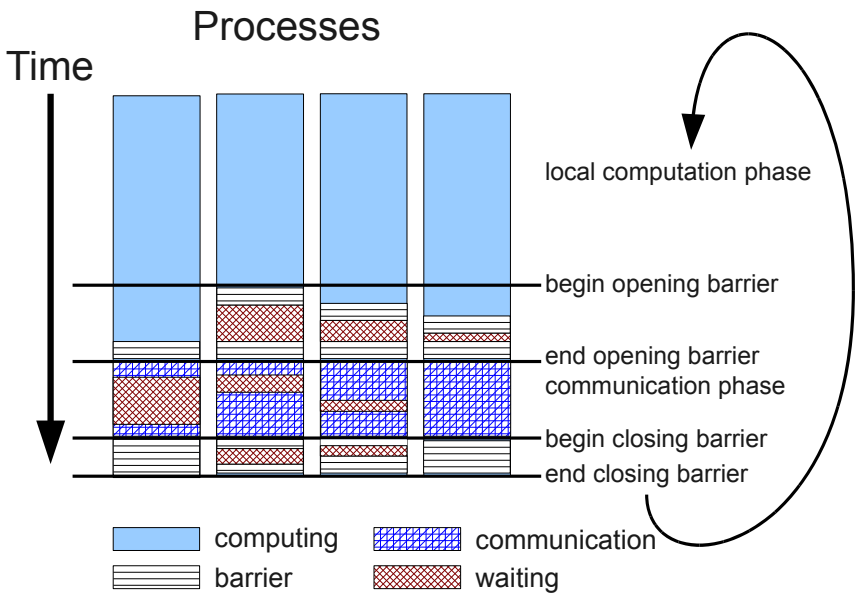

Figure 2.1: Simplified view of HPC applications that follow the bulk-synchronous SPMD model. All processes cycle through alternating computation and synchronization/-communication phases. Adapted from [6] and [45].

Petrini, Kerbyson, and Pakin have analyzed parallel application performance on a supercomputer with 8192 processors [67]. They have identified and solved a performance problem and achieved a factor of 2 speedup with modifying the actual parallel application. They observed in detail, how OS background activity on different nodes of the supercomputer affected performance. Another important observation they made, is that micro-benchmark results of elementary operations need not correlate with application performance, even when the application makes heavy use of the respective operation. So, evaluating an HPC system for parallel application performance cannot rely on micro-benchmarks alone.

To avoid being mislead by inappropriate benchmarks, large institutions with specific computing requirements often publish their own benchmark suites that mimic the behavior of typical applications they are using. Examples are the NAS Parallel Benchmark of the NASA Advanced Supercomputing (NAS) Division and the Sequoia Benchmark Codes [10] of the US Advanced Simulation and Computing Program.

Tsafrir and associates [89] have presented a probabilistic model for the effects of OS background activity (system noise). From this model, they deduced that the scaled OS noise — that is, the fraction of total CPU time used by OS background activity — should be in the order of 10^{-5} to minimize the influence on application performance for clusters with thousands of nodes. They identified timer ticks as the main cause of OS noise, as experienced from an application, and propose to avoid unnecessary ticks.

Ferreira, Bridges, and Brightwell have analyzed the influence of different types of OS background activity on different parallel applications [27]. They create artificial background activ-

ity, vary its characteristics in terms of frequency and duration of each activity period, and measure and compare the influence on the different applications. They observed that low-frequency high-duration noise caused much higher performance degradations than high-frequency short-duration noise, up to a slowdown by a factor of 20 from only 2.5% of CPU time consumed by low-frequency background activity. As a result, they state that considering OS noise benchmarks alone does not suffice to assess application performance — an advice that we shall follow in our evaluation in Section 4.2.1 on page 60.

2.6 High-Performance Computing on General-Purpose Clouds

In the previous section, we have introduced the bulk-synchronous SPMD application model and considered performance analysis in the traditional supercomputer and cluster context, areas intentionally designed for HPC workload. The widespread availability of commercial cloud offerings, such as Amazon EC2, has led to much research on the performance of HPC workloads on general-purpose clouds, which have been designed for server workloads. In this section, we present several studies that focus on different types of computational scientific workloads. We did not restrict the selection to classical HPC applications (the bulk-synchronous SPMD-type), because evaluations with other types of workloads still provide interesting insight about contemporary clouds.

Napper and Bientinesi [63] explicitly posed the question whether a cloud computing provider can reach the TOP500 supercomputer list [60]. For this purpose, they evaluated the performance of the high-performance linpack benchmark, which is used to rate supercomputer systems for the TOP500 list, on Amazon EC2. Despite acceptable virtualization overhead on a single node, Napper and Bientinesi observed a drastic performance loss when scaling to two nodes (even worse with more nodes). They attributed the lack of scalability to the low performance of the network employed in the Amazon EC2 cloud.

Montero and associates have proposed to virtualize high throughput computing clusters using cloud environments [61]. They have evaluated the influence of using virtual nodes from a remote cloud provider (Amazon EC2) in addition to an existing cluster and observed acceptable results. Further, they proposed a performance model for capacity planning. Their evaluation focuses on jobs that run independent on each node without communication interdependencies. As a results, the high communication latencies they observed (inside the cloud and between the cloud and the existing cluster) only influence job startup time, when input data must be transfered.

Juve and colleagues have evaluated scientific workflow applications on Amazon EC2 and compared performance and costs with an HPC system [46]. Scientific workflow applications are sets of loosely-coupled tasks that communicate via files that tasks read/write as their respective input/output. The authors of [46] restricted their analysis to running workflows on one node. They observed virtualization overheads of less than 10 %, yet found the main bottleneck of current clouds in the low performance of disk I/O.

Iosup and associates have studied a similar class of applications, which they call many-task scientific workloads [41]. One of their main contributions is a detailed analysis of traces from real clusters and compute grids, on which they base a performance analysis of existing compute clouds. They found that the performance of actual scientific workloads is worse on existing compute clouds than on clusters or grids by about an order of magnitude. However, compute clouds deliver performance at much lower costs and provide an interesting alternative for temporary short-term demands for compute resources. The authors also refer to [40], where they analyzed grid workloads and found that most parallel compute jobs employ only up to 128 processors.

2.7 Related Work

In this section, we present some publications that are directly related to our work. To our current knowledge, there is no previous work that combines cloud and cluster network management to provide virtual clusters with access to distinct cluster networks, and at the same manages isolation in the cluster network. There has been much research on providing virtual clusters however, and there are some known designs that advance datacenter networks by introducing traditional cluster interconnect topologies or by providing configurable virtual network topologies with cloud-like interfaces and automatic configuration.

We also regard the palacios hypervisor as related work, a lightweight hypervisor specifically designed for HPC, in contrast to the general-purpose combination of Linux and KVM, which we analyze in this work.

2.7.1 Virtual Clusters

The idea to use virtual clusters for scientific compute workloads has formed in the era of grid computing already, long before cloud computing existed as a term. Foster and colleagues analyzed grid workloads on virtual clusters in [31], a project which eventually led to the Nimbus cloud toolkit [47, 70]. In a more recent work, Marshall, Keahey, and Freeman propose to dynamically extend existing clusters with nodes from a compute cloud [56], depending on demand, thereby making existing clusters elastic. They describe logistical problems that arise, present a resource manager based on Nimbus for extending existing clusters with nodes from a compute cloud, and discuss three resource allocation policies for this purpose.

Nimbus provides users with virtual clusters out of the box. It supports only TCP/IP networking on Ethernet however, with basic network isolation, whereas we explicitly target distinct cluster networks with automatic configuration of network isolation. Nimbus provides a network connection between the nodes of a virtual cluster without further configuration options, whereas we strive for a rich set of management features a user may apply to his share of the network.

We shall discuss the nimbus toolkit in Section 3.4.1 on page 49 as one of the existing cloud management frameworks. Our prototype is based on the OpenNebula cloud management framework [65], even though the extensibility concepts of nimbus are sufficiently similar that our approach can be transfered.

The OpenNebula cloud management framework itself does not support virtual clusters, but considers each VM a separate entity. Yet, Anedda and colleagues have developed an extension to OpenNebula (called virtual cluster tools) that allows to trigger operations, such as deployment, on complete virtual clusters. Their focus is however to achieve save and restore operations (suspending execution and saving a VM's state to a file) on all VMs that comprise a virtual cluster at the same time [4]. That way, they avoid to disturb parallel applications running inside the VMs.

Like the nimbus toolkit, the virtual cluster tools of Anedda and colleagues do not regard cluster networks, such as InfiniBand. They consider only MapReduce parallel applications, not classical HPC applications in the bulk-synchronous SPMD model.

Huang, Liu, and Panda developed a batch job management system that runs each HPC job in a VM that provides the required environment for the job [37]. That way, they use an existing (physical) cluster with virtualization. In our approach, we aim at providing a user with the view of his own cluster that runs itself virtualized, as it is the case with the nimbus toolkit and the virtual cluster tools for OpenNebula. That way, a user can employ a custom job management system and different virtual clusters are intrinsically more separated than the jobs in the system of Huang, Liu, and Panda.

2.7.2 Datacenter Network Virtualization

Greenberg and colleagues have introduced a novel architecture for data center networks, which they call VL2 [34]. They aim for bandwidth that is independent from the location of a server in the physical topology. They employ the Clos network topology typically used in cluster interconnects and replace Ethernet's spanning tree protocol with a combination of customized switching and layer-3 routing. They introduce an IP address translation directory to separate IP addresses used to name services from those used in routing. VL2 increases the bandwidth available in datacenter networks and makes IP addresses independent from the routing topology, at the price of introducing an IP address translation mechanism to the TCP/IP stacks of each server node. VL2 clearly aims at server workloads in TCP/IP networks based on Ethernet and therefore focuses on achieving high bandwidth, not latency, as is the case in cluster networks. It is intrinsically dependent on TCP/IP, because layer-3 routing based on IP-addresses is an essential building block of VL2.

Benson and associates have proposed the cloud networking system EPIC that allows users far-reaching control over the network connectivity of their VMs in compute clouds [14]. Users can define virtual networks and thereby configure custom isolation and connection schemes for VMs. In addition, EPIC allows to configure transparent middleboxes, such as firewalls or intrusion detection systems. In contrast to traditional network infrastructure, EPIC implements the virtual network specification submitted by a user automatically as a part of cloud management, without the need for human interaction.

EPIC [14] virtualizes datacenter networks and offers automated configurability options that did not exist for such networks before. It is specifically designed for such networks however, regarding the goals the authors have set and the mechanisms they employed, and therefore cannot be transfered to cluster networks. For example, firewall middleboxes are not practical in

cluster networks because of the latency they add. In our work, we employ the mechanisms of cluster networks for isolation and virtualize the existing configuration interfaces of the cluster network. In contrast to datacenter networks, keeping native performance, especially network latency, is the primary goal.

2.7.3 Lightweight Hypervisors

In our work, we analyze how well KVM on Linux, a hypervisor typically used for server workloads on a commodity OS, serves as a virtualization layer for HPC workload. The palacios hypervisor follows an alternative approach [50]. It can be embedded into the kitten lightweight kernel, which aims at providing applications with a low-overhead environment, and provides itself a slim virtualization layer aimed at incurring low overhead. Both palacios and kitten aim at maximizing the resources allocated to applications by reducing background activity by the lightweight kernel or virtualization layer. In combination, they allow to run commodity OSs, such as Linux, in a virtualized environment, and still provide custom applications with kitten's low overhead environment.

Palacios is limited however to single-CPU guests. So, a multi-core host can only be utilized with virtualization as several independent virtual single-core nodes. Palacios provides PCI-passthrough device access for Intel Gigabit Ethernet adapters, Mellanox InfiniBand HCAs, and SeaStar, an interconnect used by Cray. However, PCI-passthrough alone allows only one VM to access a device. While palacios and kitten are interesting research projects, they impose limitations for practical use with multi-core hosts and several VMs that employ high-speed interconnects. In contrast, our approach of employing Linux and KVM, leads to a readily usable solution.

3 Approach

This chapter presents our approach towards virtual InfiniBand clusters. We begin with an introduction to our overall architecture, which is divided into three areas: node virtualization, network virtualization, and cloud management. Node and network virtualization provide the mechanisms that cloud management eventually orchestrates to provide virtual clusters. We reflect this relationship in the structure of this chapter: First, we present an overview of our architecture in Section 3.1. Second, we describe how we virtualize physical compute nodes to provide virtual nodes in Section 3.2 on page 20. Third, we discuss how we virtualize the cluster network and a user's view of it in Section 3.3 on page 30. Finally, we briefly describe how a cloud management framework feasible for high-performance computing (HPC) employs node and network virtualization to provide virtual HPC clusters in Section 3.4 on page 46.

3.1 Basic Architecture

Our goal is to transform a physical HPC cluster, a collection of compute nodes connected by a high-performance cluster interconnect, into a cloud computing infrastructure for HPC workloads. To reach this goal, we employ virtualization of compute nodes and virtualization of the cluster interconnect, in our case InfiniBand, to offer dynamically and automatically allocated virtual clusters. In contrast to physical clusters, our virtual clusters can dynamically grow or shrink at runtime, a property commonly called elasticity in cloud computing.

We provide each virtual cluster the impression of using the InfiniBand interconnect by itself, albeit with reduced bandwidth. The advanced configurability options of InfiniBand (compared to contemporary Ethernet) can be used in a virtual cluster's view on the network, using existing management tools, and without an adverse influence on network performance. In contrast to physical InfiniBand hardware, our virtualized cluster interconnect offers the advantage that a user can revert to a default setup provided by the HPC cloud, when he does not want to hand-tune the configuration. Figure 3.1 on page 18 provides a schematic overview of the HPC cloud architecture we target.

Compared to commercial server clouds, HPC applications make much stronger demands on quality of service (QoS). An HPC cloud has to allocate resources in a way that one can give guarantees for the performance of processes on individual nodes and for the QoS a virtual cluster does experience when using the cluster interconnect. Despite virtualization, the performance characteristics of the cluster interconnect, such as low-latency, low-overhead communication, have to be maintained.

Our approach is early work in the area of incorporating cluster interconnects into cloud computing for HPC applications. Previous work on virtualizing InfiniBand (or other cluster interconnects) centers on virtualizing the InfiniBand host channel adapter (HCA) to provide virtual

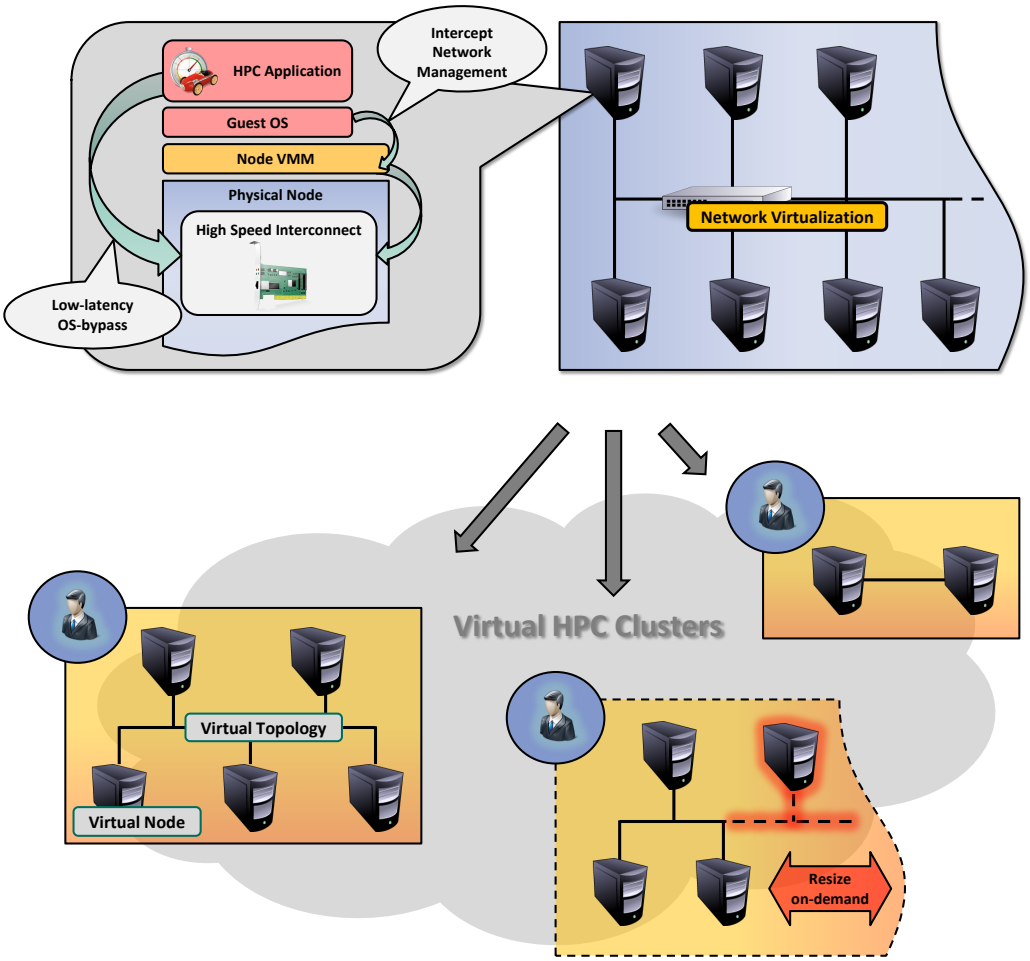

Figure 3.1: Basic architecture of our approach. Node and network virtualization, controlled by an appropriate cloud management framework, provide elastic virtual InfiniBand clusters on top of a physical cluster.

machines (VMs) access to the interconnect [53], but leaves out aspects such as a VM's view of the network and isolation in the presence of several VMs accessing one HCA. Former studies on the performance of HPC applications in compute clouds cover many different types of scientific applications [41,46,61], but restrict themselves to existing contemporary cloud computing infrastructures, which employ Ethernet networks and do not offer cluster interconnects.

We propose to employ Linux and the Kernel-based Virtual Machine (KVM) as well as the existing InfiniBand driver stack in Linux as the virtualization basis for building our HPC cloud architecture. We follow the question, whether this commodity hypervisor and the existing InfiniBand system software can be transformed to support such an architecture. Linux and KVM have some practical advantages over alternatives, however, HPC applications make much stronger demands on QoS, regarding the allocation of CPU and network resources, than server workloads.

We are currently focusing on the InfiniBand system interconnect, but expect that our findings can be transferred to other cluster interconnects or future Ethernet generations. Current cluster interconnects already share many ideas and concepts, such as OS-bypass and transport layer protocol offload[1], and the evolution of Ethernet also does begin to incorporate these techniques[2].

In the course of this thesis we present an analysis and a first prototype. In each of the following sections, which are node virtualization, network and topology virtualization, and HPC cloud management, we begin with an analysis of existing mechanisms. Then, we discuss how we can transfer them to a virtualization environment for HPC workload, and present the design of our prototype.

We follow a similar structure in the remainder of this section: We state the type of HPC applications we consider in this work, discuss some practical advantages of Linux and KVM over alternatives, and present an overview of our prototype.

We use the bulk-synchronous SPMD model to characterize the designated workload of our HPC cloud. We have already introduced this application model and presented references to former work on the performance characteristics of such applications in Section 2.5 on page 10.

In our work, we follow the question, whether Linux and KVM can be transformed to serve as a virtualization layer appropriate for such applications. Linux and KVM have practical advantages over alternatives, mainly because of their widespread use.

Existing management frameworks for cloud computing, such as OpenNebula, support KVM, amongst other commodity hypervisors. Therefore, we can employ the existing host, hypervisor, and VM management software stack in OpenNebula.

Considering the practical use of an HPC cloud, the reliability of the system software has a major influence on the reliability of the provided virtual clusters. Compared to the scenario of native HPC systems (see [78]), HPC clouds add another software layer for virtualization, and thereby another potential source of errors. We prefer Linux and KVM over alternative virtualization stacks, because they have the advantage of a large user base, which contributes to excessive testing, eventually leading to improved reliability. In addition, commercial support is available for Linux, so it is possible to get bugs and problems that occur in a production environment solved in a timely manner. In contrast, research OS kernels aimed at HPC workload are primarily developed to experiment with new ideas, not for long-term, large-scale production use.

Further, Mellanox, the manufacturer of the InfiniBand host channel adapters (HCA) we employ in our prototype, will release drivers with SR-IOV virtualization support (see Section 3.2.1 on page 24), for use with Linux and KVM, in a few weeks. Adapting these drivers for use with another virtualization layer would require porting work and continuous maintenance of another software artifact.

[1] As an example, see [62], which introduces Myrinet from a system architecture perspective and proposes a paravirtualization concept comparable to one for InfiniBand [53].

[2] The Internet Wide Area RDMA Protocol stack (iWARP) [74] features RDMA and the direct placement of received data in an applications address space by iWARP-capable network interface cards.

Our prototype comprises three areas: We virtualize each node of a physical cluster, we virtualize the cluster interconnect, and we orchestrate node and network virtualization with an HPC cloud management.

On each physical node, we employ the KVM hypervisor on Linux, a combination which has so far been used primarily to virtualize server and commercial workloads. We customize the kernel configuration of the host OS and disable unnecessary features — for example, support for swapping, which is not needed and undesired for HPC applications (we shall evaluate the influence of different kernel configurations on OS noise and application performance in Section 4.2.1 on page 60). We shall name a VM intended for HPC workload a virtual node — many virtual nodes form a virtual cluster.

We employ the mechanisms InfiniBand provides for network isolation (partitions) and to guarantee QoS (virtual lanes), and extend them to implement isolation and QoS effectively in a virtualized environment. In addition, we provide a virtual cluster with the impression of its own InfiniBand network, which allows a user to configure his own partitions and QoS policies within his share of the network.

We extend the cloud management framework OpenNebula to automatically configure and deploy virtual clusters and their network partitions.

3.2 Node Virtualization

In this section, we turn the physical nodes of a cluster into virtualized environments that provide several virtual cluster nodes capable of running parallel HPC applications. For this purpose, we employ an existing hypervisor for commercial and server workloads, KVM on Linux, and configure it according to our primary focal points — that is, we aim for low virtualization overhead, and even more important, minimal background activity in the hypervisor and in the host OS.

All virtual nodes and the host require using the cluster interconnect. The virtual nodes utilize it for synchronization and communication, and the host OS employs the interconnect for management traffic, in place of a separate, additional network infrastructure. However, we expect that a physical node will only have one cluster interconnect HCA (or potentially two for redundancy), but hosts more virtual nodes. In summary, we have to provide shared access to the HCA for several OS instances, running in VMs and on the host system.

Our virtual HPC environment has to maintain the performance characteristics of the cluster interconnect, especially low latency, because they are key to parallel application performance. Server workloads typically process several requests at a time. During blocking times of one task (e.g., while it is waiting for a database query to complete), there are other tasks ready to run. In contrast, HPC applications typically work on only one computational problem. When the tasks comprising the parallel application reach a point in execution where they need to exchange data or synchronize with each other, there is no other work that can be done while waiting for communication. As a result, increasing communication latency by virtualization may be acceptable for server workloads, because it does not harm overall throughput. However, for

HPC workloads, increased latency directly results in longer waiting times and often, depending on the actual application, reduced application performance.

This section comprises two parts: First, we examine existing components, the KVM hypervisor and InfiniBand HCAs, that we use in our approach to provide virtual compute nodes on a physical host, and present some aspects of contemporary HPC cluster nodes that we consider in virtualizing them. Second, we describe how we employ and configure these components to fulfill our goals. Thereby we transform a Linux/KVM virtualization environment targeting server workload, such that it is capable of hosting virtual nodes of an HPC cluster.

3.2.1 Analysis

Server Virtualization with Linux/KVM

Research in virtualization has led to the construction of a variety of hypervisors, ranging from prototypes to commercial products and employing virtualization techniques such as binary translation or special CPU execution modes. One such hypervisor is the Kernel Based Virtual Machine (KVM, [49]), which adds support for the current x86 virtualization extensions Intel VT [90] and AMD-V [2] to the Linux kernel. KVM exports these mechanisms via the Unix file abstraction, so one can implement a hypervisor based on KVM completely in user-space. The QEMU emulator [13, 71], originally a full system emulator based on binary translation, has been modified to complete KVM to a full type-2 hypervisor. All interaction between KVM and QEMU, such as the setup of virtual CPUs or the configuration of which instructions shall cause a VM exit (a trap from the guest back to the hypervisor), uses ioctl and read/write system calls.

CPU Virtualization VMs based on KVM are scheduled like other regular Linux processes. Guest code is executed in the QEMU process when it enters guest mode, introduced with KVM in addition to user and kernel mode. Besides the default scheduling policy, the completely fair scheduler (CFS), which employs dynamic priorities and was designed with interactive processes in mind, Linux supports real-time policies with static priorities and an idle priority class that is scheduled only when the CPU would otherwise be left idle [57].

QEMU supports multiprocessor guests using POSIX threads. Each logical CPU is implemented by a separate thread, so SMP guests can run parallelized on SMP hosts. Linux scheduling supports CPU affinity in a configurable way. Concretely, each thread has a CPU affinity mask, a bitmap that defines on which CPUs a process can be dispatched. By default, a thread can be scheduled on all CPUs, with the affinity mask set to all ones.

Memory Virtualization Guest physical memory is built from parts of QEMU's virtual address space, called memory slots. QEMU[3] uses an *ioctl* system call to create memory slots and

[3] We describe the mechanisms provided by KVM using the example of QEMU. Of course, they work the same way with every other hypervisor that employs KVM.

specifies for each slot its size, its designated address in guest physical memory, and the region in QEMU's virtual address space to be mapped to the guest.

Contemporary Cloud Computing and virtualization setups employ various techniques to overcommit memory — that is, to assign more memory to guests (as guest physical memory) than there is physically present in the host system (see the Related Work Section in [32] for a recent overview). Memory overcommitment allows to host more VMs per physical host than if physical memory was just partitioned, and helps to increase the utilization of physical resources [5].

Device Virtualization QEMU provides a full set of virtual devices, such as virtual disks based on disk image files or physical storage devices, and virtual Ethernet NICs, which can be connected to the Linux network stack. QEMU emulates basic mechanisms such as I/O memory and I/O ports, and peripheral buses like PCI and USB for guests.

KVM and QEMU support PCI passthrough, also referred to as PCI direct device assignment [97], to give a guest OS access to a physical PCI device. To maintain isolation, the hypervisor must confine DMA operations of the passed-through device to the memory of the guest, requiring an IOMMU. See Section 2.3 on page 8 for more details on PCI passthrough with IOMMUs. Depending on the policy used for the memory mapping in the IOMMU, the virtual machine's memory must be pinned to the host physical memory [95, 97]. I/O Memory regions of the PCI device are memory-mapped to the guest[4]. Interrupts are delivered to the host and forwarded to the guest by KVM. When an interrupt occurs while a guest is running, execution always returns to the hypervisor (called a VM exit), even when the interrupt is designated for the guest currently running. If a processor could deliver certain interrupts directly to a running guest, the overhead and latency imposed by the return to the hypervisor (called a VM exit) would be avoided. However, current versions of Intel VT and AMD-V only allow the indirect delivery path. Access to the PCI configuration space is emulated by QEMU. With devices supporting Single Root I/O-Virtualization (SR-IOV) — that is, the device itself implements virtualization and offers several virtual interfaces (see Section 2.3 on page 8) — PCI passthrough is used to give a guest access to one of the virtual functions of the device.

Access to Cluster Interconnects

In contrast to conventional network technologies, cluster interconnects like InfiniBand or Myrinet rely on protocol offload and OS-bypass (user applications can directly access the HCA) to achieve high bandwidth and low latency[5]. Usually, these HCAs implement a complete, self-sustained network protocol stack up to transport layer of the ISO/OSI model.

We use ConnectX-2 InfiniBand HCAs from Mellanox in our prototype cluster. Virtualizing these devices requires us to consider their specifics. We briefly describe the interface these

[4] The Linux kernel exports PCI resources via the sysfs virtual filesystem. Thus, QEMU can mmap device memory and hand the mapping on to the guest. See Documentation/filesystems/sysfs-pci.txt in the Linux kernel sources for details.

[5] This work is focused on InfiniBand. For an introduction to Myrinet from the perspective of system software see [62] about Myrinet virtualization.

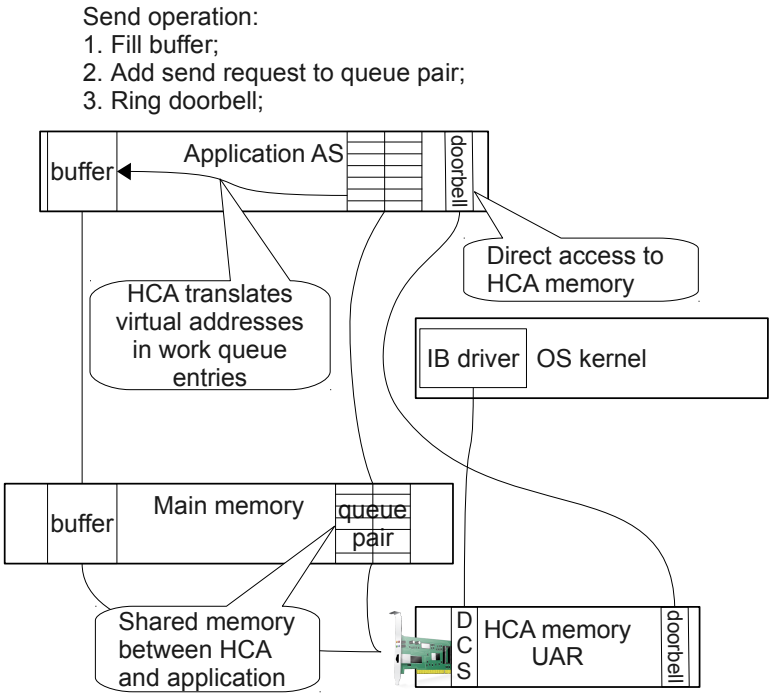

Figure 3.2: The ConnectX-2 software interface with OS-bypass.

adapters provide to system and application software[6] and which options exist to access such HCAs in a virtualized environment.

Mellanox ConnectX-2 HCA The ConnectX-2 HCAs connect to the PCI Express bus and export two I/O memory regions, the device control section (DCS) and the user accessible region (UAR)[7]. The DCS contains a register-based command interface that the device driver uses to manage and configure the various abstractions and mechanisms of the InfiniBand architecture, like queue pairs, event queues, memory registrations, etc. The UAR is split into several regions each 4 KB in size, matching the smallest page size of most architectures with PCI express buses. Each application using InfiniBand gets one page of the UAR mapped into its virtual address space.

OS-Bypass Concept Once a transport layer connection has been established, using the conventional way of posting requests to the device driver via system calls, direct access from user-space (OS-bypass), works as follows: A user application can command send and receive operations by adding a work queue entry to the matching send or receive queue, a memory buffer shared with the InfiniBand HCA. In the send case, the actual operation is triggered by

[6] The description of the software interface is derived from the Linux device driver (drivers/infiniband/hw/mlx4/ and drivers/net/mlx4 in the kernel source tree) and the device-specific part of the user level verbs library libmlx4.

[7] We deduced the long form of DCS from its function because it is not mentioned in publicly available documents. The long form of UAR occurs in various publications, however.

writing to a doorbell register in the UAR page. In the receive case, the application can poll the completion queue associated with the receive queue, while interrupts are disabled for this completion queue. See Figure 3.2 on page 23 for an illustration of the interface of the HCA and the OS-bypass mechanism.

HCA Virtualization Our focus on low latency rules out the conventional network virtualization approaches that are employed with Ethernet, where all network traffic flows through the protocol stack of the guest OS and, additionally, through that of the host OS or a device driver guest like in Xen [12] (see Section 2.3 on page 8). They aim for high throughput where an additional delay of single packets does not matter much (see for example [18, 58, 76] that all focus on throughput measurements). Liu and associates have proposed VMM-bypass I/O in [53], an extension to OS-bypass that allows an application to circumvent all layers of system software, OS and hypervisor, when accessing the HCA. They have shown that an application using VMM-bypass from within a VM suffers only a small overhead compared to an application on the host OS using OS-bypass. We follow the VMM-bypass approach for virtualizing cluster nodes.

In the following, we list the three virtualization concepts appropriate for InfiniBand HCAs that we are aware of (we shall choose one of them in Section 3.2.2 on page 29):

PCI passthrough to dedicate the device to one VM. This concept is usable with KVM on Linux, but allows only one HPC VM per HCA and requires an IOMMU, see Section 3.2.1 on page 22. In addition, the host OS requires a separate network connection — for example, an additional InfiniBand HCA or a complete Ethernet infrastructure. The guest OS employs the regular InfiniBand driver stack. Dedicated HCAs are more expensive and less flexible than shared HCAs. The number of HPC VMs per host is restricted to the number of InfiniBand adapters built into this node.

Paravirtualization as shown for a former hardware generation in [53] and [36]. The approach can be transferred to current hardware. It does not need an IOMMU. However, it requires a modified device driver in the host as well as in the guest. These paravirtualized drivers require maintenance and have to be adapted when OS, hypervisor, or hardware change. As an example, the available source code of the prototype presented in [53] targets only outdated versions of Xen and Linux, and supports only one guest.

SR-IOV for virtualization by the device itself (see Section 2.3 on page 8). The device offers several virtual interfaces that behave like self-contained PCI devices, called virtual functions. Several guests can each get access to one virtual function on its own via PCI passthrough. Device drivers and firmware patches for SR-IOV are developed and provided by Mellanox, the manufacturer of the HCAs. We expect a high quality of the drivers because of the manufacturer's commercial interests.

These concepts are all known to work with Mellanox ConnectX-2 HCAs. Mellanox will release drivers with SR-IOV virtualization support, for use with Linux and KVM, in a few weeks. PCI passthrough and SR-IOV can be applied to other types of InfiniBand HCAs as well: PCI

passthrough works, unless the HCA does not conform to PCI standards (compare [96]); SR-IOV depends on a manufacturer to add appropriate functionality to the HCAs and provide specifications and/or drivers for SR-IOV access.

Paravirtualization of an InfiniBand HCA with VMM-bypass, however, is specific to the software interface of an actual HCA. Buffers and request queues as shared memory between HCA and application are inherent to the InfiniBand architecture, but actual HCAs differ in how an application triggers the HCA to process new queue entries. This mechanism is the main challenge for the paravirtualization of other HCAs. In the case of ConnectX-2 HCAs, a hypervisor must provide guest applications direct access to an UAR page to enable VMM-bypass access (see [53]). For this purpose, it maps these I/O memory pages to the VM's guest physical memory. The necessary basic OS abstraction and mechanism, virtual address spaces and memory mapping, are already in place, because a hypervisor already utilizes them to implement guest physical and guest physical memory, and an OS running natively employs them to implement OS-bypass.

HCA Resource Sharing HPC applications access high-performance interconnects like InfiniBand directly, using operating system and, in virtualized environments, one of the aforementioned approaches to HCA virtualization. Neither the guest OS nor the hypervisor can directly supervise how applications use the interconnect or determine whether its bandwidth is shared fairly between all applications (compare the discussion of an indirect approach in [73]).

We assume that the resources of the InfiniBand HCA itself do not pose a bottleneck when using the full bandwidth of its ports. Otherwise, fair allocation of interconnect resources would demand a fine-grained control of how many queue pairs, memory registrations, or buffer queue entries, amongst other things, a VM is permitted to use.

An application can request a HCA to generate interrupts on events like the reception of data. With improper interrupt routing, it may thereby cause overhead for other VMs. The situation can be improved when the HCA triggers separate interrupts, depending on which guest is to be notified. That way, the individual interrupts can be routed to the CPUs where the respective VM is running, and other guests remain undisturbed.

Contemporary HPC Cluster Nodes

Previous studying how operating systems affect HPC performance and scalability suggests that OS background system activity on the individual compute nodes must be kept minimal (see Section 2.5 on page 10). Preferably, it occurs simultaneously on all nodes to avoid chains of process interruptions, which can dramatically reduce application performance because they potentially block all nodes from making progress.

In Cloud Computing for commercial and server workload, many VMs are multiplexed to a single physical CPU using timesharing, without considering the applications running inside a VM. HPC Clusters usually employ job schedulers however, also called Job Management Systems [25], which aim to maximize processor utilization without harming the performance of individual parallel applications. These schedulers incorporate sophisticated algorithms, based

on rich practical experience [26]. In contrast to Cloud Computing, cluster job schedulers often dedicate CPUs to a single process or use coordinated timesharing algorithms like gang scheduling [26, 66].

In the remainder of this section, we shall utilize the existing virtualization techniques discussed above and configure them appropriately to provide a virtualized environment feasible for HPC.

3.2.2 Transformation for HPC Clouds

We have reviewed the Linux/KVM virtualization environment and its configuration options (e.g., Linux scheduling policies), and InfiniBand HCAs and the possibilities to virtualize them in the previous section. Here, we shall employ and combine the KVM hypervisor with InfiniBand HCA virtualization to reach our goals for node virtualization — that is, to turn a physical cluster node into a virtualized environment that provides virtual cluster nodes with low-overhead access to the InfiniBand interconnect and minimizes perturbation from background activity.

CPU Allocation

Policy We dedicate a physical CPU core to each logical CPU of a virtual cluster node. We expect that users of virtual clusters will run job schedulers that aim to maximize utilization of (in this case virtual) processors, so we do not duplicate this attempt at the level of VMs and explicitly exclude timesharing between several VMs with HPC workload on one physical CPU.

We decided to dedicate cores based on three reasons: First, job schedulers for parallel applications do exist, ready to be used. They have more a-priori knowledge about a job than the underlying virtualization layer, because they can rely on the process abstraction in the guest OS, in contrast to the virtualization layer. Second, timesharing between VMs makes execution time, as experienced by the guest OS, nonlinear and nondeterministic. As a result, jitter is induced onto the execution speed of the individual processes that form a parallel HPC application. Numerous surveys show that the performance of parallel HPC applications can suffer badly when execution time of iterations differs between nodes, because a few late processes block all others in global synchronization operations (see Section 2.5 on page 10). Third, present and future many-core architectures provide enough CPU cores. These architectures shift the focus from sharing limited compute resources between several tasks towards utilizing many cores in complex memory hierarchies and communication structures in an efficient way [7, 8].

Cluster job schedulers offer existing and readily usable expertise in parallel job scheduling. Applying timesharing to VMs, without further knowledge on the jobs they are running, can have an adverse effect on the performance of the individual applications. The hypervisor does not know whether the HPC application inside a VM is making actual progress or it is polling for synchronization messages from other nodes. In both cases, it fully utilizes the allocated CPU time. Clearly, a VM that is scheduled for polling can prevent other VMs from making progress, so neither the scheduled VM nor other VMs make use of the CPU time. The other way round,

even when the hypervisor would discern between VMs in polling wait and VMs ready to make progress, polling for communication events means that these events are configured not to trigger interrupts. As a result, after receiving a message, a VM and the hypervisor will only learn about the transition from polling wait to ready to make progress, when the VM is scheduled again. In the meantime, a whole timeslice can be wasted polling in another VM. Such local delays in one node will increase the time the other nodes have to wait (also polling), and increase the time wasted in polling wait, there. Instead of improving utilization, time sharing would even reduce overall efficiency because more CPU time is spent polling. As a consequence, we dedicate each logical CPU core of a HPC VM a physical CPU core and delegate responsibility to utilize the CPU core for HPC workload to the scheduler running inside the virtual cluster.

As a downside of node dedication, a physical CPU core stays idle, when the associated virtual CPU is idle, reducing overall system utilization. It happens in the time interval between one job finishing and the next being started by the cluster scheduler and when a cluster scheduler also dedicates a (virtual) core to a single application and this application waits (blocking) for communication or synchronization operations[8]. We propose to utilize these cycles for non-time-critical, non-HPC jobs, scheduled independently from the HPC VMs with strictly lower priority. This idea is derived from [86], where Stiehr and Chamberlain proposed to introduce a low-priority band below the regular dynamic priorities in the Linux kernel to increase the utilization of cluster nodes by running background jobs, which do not influence the progress of the primary workload[9]. Blocking communication operations generate interrupts upon completion anyway and thus allow a seamless continuation of an HPC job at the point it becomes ready again.

When a HPC application polls for the reception of synchronization messages, it can donate the waiting time by changing the polling to a blocking wait after some threshold time. It requires that the time of waiting can be expected to be long enough to justify the context switch and the overhead of interrupt handling and switching back to the HPC application. See [15] for a discussion of optimal threshold times.

Mapping to Linux Scheduling We have discussed our policy of allocating physical CPU resources to virtual CPUs in the section above. We dedicate a physical CPU core to each virtual CPU for HPC workload and add low-priority non-HPC background jobs to increase utilization when HPC workload leaves virtual CPUs idle. In the KVM hypervisor we employ on a Linux host OS, a guest is executed as part of its hypervisor process (compare Section 3.2.1 on page 21), so VMs are scheduled like regular processes. Therefore, we implement our CPU allocation policy by mapping it to the Linux scheduling policy and assigning appropriate priorities to the hypervisor processes.

We employ CPU affinity masks to implement core dedication (running each logical CPU of a HPC VM on a separate physical core). They allow to confine each HPC hypervisor process, to

[8] Early experiments with the High Performance Linpack benchmark have shown this behavior. In one run, progress was blocked due to communication for 2 seconds out of 120.

[9] In the mean time, the *SCHED_IDLE* scheduling class has be added to the Linux kernel and provides a way to give processes the lowest possible priority, strictly below all other processes.

a separate physical core, and thereby eliminate competition for CPU resources between HPC VMs, because the guest software stack runs inside the hypervisor process.

We run low-priority background VMs in addition to the HPC VMs to consume the CPU time left idle by the HPC workload and thereby increase utilization. We cannot use the static Linux soft-real-time priorities for this purpose, because InfiniBand device drivers did not work properly in VMs with real-time priorities in our tests (see our evaluation in Section 4.2.3 on page 73). Instead, we employ the Linux-specific SCHED_IDLE scheduling policy. Processes assigned this scheduling policy are dispatched only when there is no other process ready (except from other processes with the SCHED_IDLE policy). We evaluate this approach in Section 4.2.3 on page 73.

Memory Allocation

Paging operations resulting from memory overcommitment would induce jitter to the execution time of HPC processes in the order of several milliseconds. As stated before, such jitter can reduce parallel application performance. In addition, it would leave processor cycles unusable for HPC tasks, because we do not use time-sharing for HPC workload. Thus, we avoid memory overcommitment and limit the memory we assign to virtual machines in total to less than the physical memory of the host. Thereby we eliminate paging as a source of unpredictability.

The design decision against memory overcommitment stems from our intent to minimize background activity in the host OS (as we use the type-2 hypervisor QEMU on Linux/KVM). Together with careful configuration of system tasks, we effectively eliminate the necessity for paging support in the host.

The Linux kernel can be configured at compile-time to omit this mechanism by disabling the option called "Support for paging of anonymous memory (swap)"[10]. See Section 4.2.1 on page 60 for an evaluation of the influence of different compile-time options on OS background activity.

It is appropriate, however, to use virtual memory and memory overcommitment for the non-HPC workloads that exploit the blocking times of the HPC jobs. We expect virtual memory implemented inside the non-HPC guest OSs to be sufficient for this purpose.

Many host-based memory overcommitment strategies target the scenario of many VMs per host of cloud computing for server workloads and aim to reduce the redundancy between the memory contents of different VMs (see [32]). When several VMs run the very same software stack from a Linux kernel, over libraries, possibly up to a web server, then the memory of each VM contains a copy of the read-only code sections of this software. Clearly, several VMs can share such pages with read-only code sections and thereby much physical memory can be saved. However, in our approach, we expect much smaller numbers of VMs, because we donate physical resources primarily to a few HPC VMs and only a small number of fill-in non-HPC VMs. Most of the memory will be used as data for specific HPC applications that differ between VMs. So, the number of VMs that can potentially share memory, as well as the

[10] The option can be found in the *General options* section of make menuconfig. In the textual configuration in .config it is called CONFIG_SWAP.

fraction of the memory in each VM that could be redundant with other VMs is lower in our approach than in cloud computing for server workloads. Therefore, we do not expect much potential benefit from such schemes.

Access to Cluster Interconnects

We have discussed how we assign basic resources, CPU time and memory, to a VM for HPC applications. In this section, we assign a VM access to the cluster interconnect, to complete its transformation to the virtual node of an HPC cluster.

We employ InfiniBand HCA virtualization based on SR-IOV, because we require interconnect access by several VMs and the host, which rules out to dedicate the HCA to one VM. SR-IOV, a standardized interface for virtualization, is supported by the manufacturer of the HCA's, compared to the unmaintained research prototype for paravirtualization.

To enforce a bandwidth sharing scheme (e.g., fair sharing or preference of a virtual cluster), we employ the virtual lane (VL) mechanism, which is intrinsic to InfiniBand, on the link between HCA and the switch it connects to. We use this mechanism in the complete InfiniBand network and discuss it in detail in Section 3.3 on page 30 about cluster network virtualization.

Host OS Customization

In the previous sections, we provided CPU resources, memory, and access to the cluster interconnect to a VM. We configured different Linux scheduling classes and employed device virtualization with SR-IOV, focused on the hypervisor process and the VM guest. In this section, we pay attention to the Linux host OS itself and tailor it to the sole purpose of hosting VMs for HPC applications.

We employ a custom-compiled host OS kernel with a custom set of compiled-in features, to reduce OS background activity. Many features are unnecessary for HPC workloads, such as swapping (compare Section 3.2.2 on page 28), and can be omitted from the kernel. In Section 4.2.1 on page 60 we evaluate the influence of different kernel configurations on OS noise and application performance.

Most Linux distributions target the application as a server or workstation OS and contain much functionality that we do not need. As a first step we disable all background system tasks and host OS mechanisms that are not absolutely necessary for managing the VMs running on the host. By default, many Linux distributions enable unnecessary daemons such as a mail server, which we do not need at all on compute nodes, and *crond* and *atd* for regular maintenance tasks.

The daemons *atd* and *crond* serve to automatically start jobs once at a time in the future *(atd)*, or on a regular basis *(crond)*. We do not need atd in the host OS of a node at all and disable it. *crond* wakes up every minute and searches a list of activities for those that need to be run at that time. The only job started by *crond* in our host OS configuration is a cleanup of system log files once a day. Clearly, most of the activity of the *cron* daemon (1439 out of 1440 wake-ups per day) is unnecessary, but it still consumes CPU time on every single node. We replace *crond*

with a central, cluster-wide job scheduler and thereby avoid the unnecessary activity of this daemon every minute on every single node (compare [45]).

3.3 Network and Topology Virtualization

We provide each virtual cluster access to the InfiniBand interconnect, isolate virtual clusters from each other, and suggest a user of a virtual cluster that his cluster was using the interconnect alone. To reach that goal, we implement isolation between network traffic of different virtual clusters (employing InfiniBand partitions). It prohibits communication beyond the borders of a virtual cluster, and prevents eavesdropping by users in foreign virtual clusters. We allocate bandwidth shares where several virtual clusters share a physical link, and establish bounds on the packet delay experienced in the presence of contention (using InfiniBand quality of service mechanisms). We employ the same mechanisms to utilize a share of the network's resources for cloud management and non-HPC traffic.

We set ourselves the target to provide each virtual cluster with a virtual view of its share of the InfiniBand network that appears like a dedicated physical network. We allow a virtual cluster to use the configurability options of InfiniBand, like in a physical network, and at the same time constrain it to a share of the network, regarding bandwidth usage and the effect of configuration. It permits a virtual cluster to use its own addressing scheme, independent from the node addresses used in the physical network or in other virtual clusters, and to customize the routing of the virtual cluster's traffic through the physical network topology to tune the routing scheme to an HPC application's communication pattern. Further, our approach allows to employ isolation inside a virtual cluster, by adding a level of recursion and splitting the virtual cluster's partition into several sub-partitions. Similar to recursive partitions, network virtualization allows to configure a QoS policy inside a virtual cluster's bandwidth share.

One of our primary goals is to maintain the performance characteristics of the interconnect, especially low latency communication (compare Section 3.1 on page 17). Therefore, virtualizing a virtual cluster's view on the interconnect may not compromise the performance of actual communication operations. We restrict the overhead of network virtualization to connection establishment and configuration operations (which operate on the virtual network view and have to be transferred to the physical network). Actual communication, such as send/receive and RDMA operations, remains unaffected for the commonly used InfiniBand transport protocols, especially that employed by MPI implementations (the exception are unreliable datagrams, which are rarely used by user applications).

After having introduced our goals above, we will present the virtual view of the interconnect that we provide to virtual clusters in greater detail in the subsequent Section 3.3.1 on page 31. We shall analyze the mechanisms InfiniBand provides for traffic isolation and to distribute network bandwidth between different network flows in Section 3.3.2 on page 33. Then, we discuss how they can be applied with virtualized access to InfiniBand HCAs and how we must extend them to ensure isolation between VMs on the same host in Section 3.3.3 on page 37. Finally, we describe how we implement virtual cluster's virtual view of the network and how we map the configuration and customization applied by a virtual cluster to the physical network, in Section 3.3.4 on page 44.

3.3.1 Virtual Network View

We provide a virtual cluster with a virtualized InfiniBand topology, so that it gets the impression of a physical network it uses exclusively. This virtual InfiniBand network supports all configuration options of InfiniBand, such as partitions for isolation, which an appropriate InfiniBand management tool (a subnet manager) started by the user can freely utilize. However, we also allow users that do not want to hand-tune the network, to rely on a reasonable default configuration. For this purpose, we differentiate two modes: A virtual cluster starts in managed mode, where a default configuration based on the physical network is active and visible, and switches to full virtualization mode, as soon as a the user starts an InfiniBand management tool (subnet manager) in the virtual cluster.

In both modes, access to the InfiniBand management interfaces by a virtual node is redirected to a state machine that simulates the virtual topology seen by the virtual cluster[11]. For this purpose, we intercept management packets in the host OS and disconnect a virtual cluster from the management interface of the physical InfiniBand network, as described in Section 3.3.3 on page 38.

InfiniBand allows to discover the topology of a network using a special type of packets. When a user runs a tool for this purpose (e.g., the command *ibnetdiscover* from the Open Fabrics Enterprise Distribution), the result will be the simulated topology which occurs to the user as the physical network. Similar, when a user starts an InfiniBand network management tool (subnet manager) in the virtual cluster, this tool's configuration requests will only affect the virtual topology and neither affect other virtual clusters nor (directly) the physical network. The network virtualization layer observes the customization done by each virtual cluster and incorporates these settings into the configuration of the physical network. We consider the following settings for incorporation into the physical configuration:

- Packet routing in switches: Management tools running in a virtual cluster can fine-tune the routing scheme to the communication pattern of an HPC application.
- Isolation: A virtual cluster can further divide its network partition and restrict possible communication relations between its nodes.
- QoS: A virtual cluster can further divide its bandwidth share for communication tasks with different priority.

The customization of the physical network for each virtual cluster is restricted to the share of network resources assigned to the respective virtual cluster. We shall describe how we map the virtual configuration to the physical subnet in Section 3.3.4 on page 44.

The simulated topology visible to a user is derived from a subset of the physical topology. It includes the hosts that run virtual nodes of the virtual cluster and all switches that can be used for reasonable routing schemes between these hosts. InfiniBand switches allow to customize packet routing and thereby choosing the routing scheme out of the options possible with the (physical) topology that best fits an application's communication pattern. By including more than only an essential selection of switches into a virtual cluster's (virtual) topology, we offer

[11] There is an InfiniBand fabric simulator called *ibsim*. It can receive management packets from several clients and generate reply packets like a real InfiniBand network would.

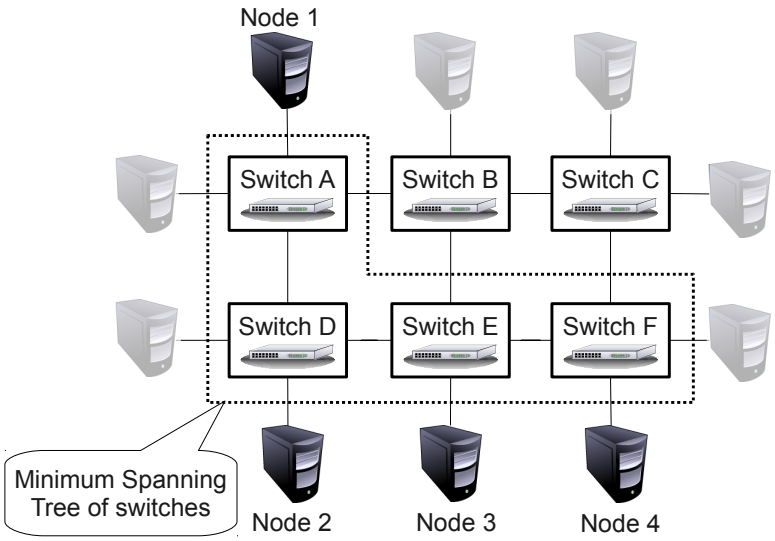

Figure 3.3: The virtual topology presented to a virtual cluster is a subset of the physical topology. A minimum spanning tree does not leave any freedom for a custom routing scheme.

a virtual cluster more degrees of freedom in setting up its routing. See Figure 3.3 on page 32 for an example. There, switches A, D, E, and F would suffice to connect the virtual cluster's nodes 1 to 4. However, adding switches B and C to the virtual topology offers the virtual cluster more paths (between nodes 1 and 3, and between nodes 1 and 4) and greater flexibility — for example, to adapt routing to the communication pattern of the currently running application.

The two modes of operation of the network virtualization layer for each virtual cluster, managed mode and full virtualization mode, differ as follows:

- A virtual cluster starts in *managed mode*. The virtual network view reflects the configuration of the physical network. In contrast to a physical InfiniBand network, where a management tool is required for a basic setup (i.e., to assign node addresses and to configure packet forwarding in switches), a virtual cluster can employ the interconnect immediately.

- The switch to *full virtualization mode* occurs, when a user operates an InfiniBand network management tool against the virtual topology. Changes to the network configuration in the virtual topology are reflected to the physical network. Customized node addresses in the virtual cluster interconnect are transparently mapped to physical node addresses in the HCA virtualization stack, so application software and InfiniBand drivers in the guest OS can use the same node addresses as assigned by the network management tool inside the virtual cluster.

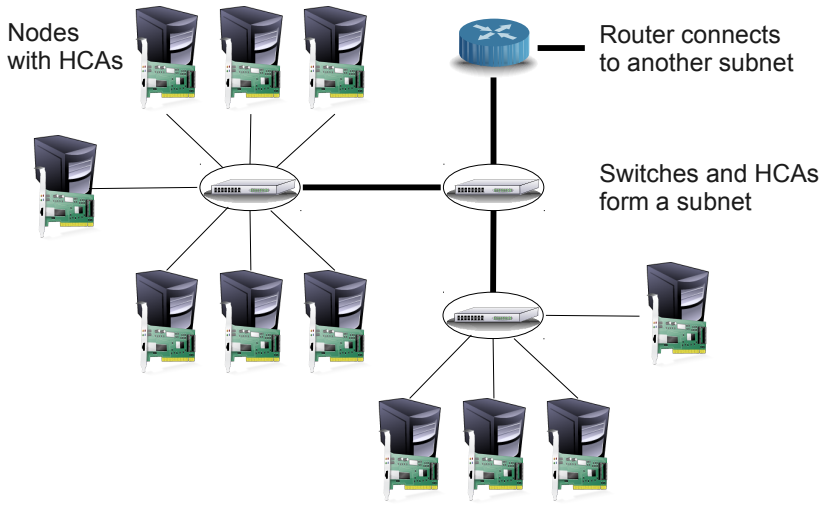

Figure 3.4: Exemplary InfiniBand network. It comprises switches, host channel adapters (HCA) that connect hosts to the network, and routers.

3.3.2 Analysis of Existing InfiniBand Mechanisms

To reach our goal of a virtual (view of an) InfiniBand network for every cluster, as described above, we employ existing mechanisms for isolation and QoS in native networks, which are defined by the InfiniBand architecture. After a brief introduction to the structure and the components of an InfiniBand network in this section, we discuss the addressing scheme of InfiniBand and its interaction with virtualization in Section 3.3.2 on page 34. Then, we describe the InfiniBand isolation mechanism (partitions) in Section 3.3.2 on page 35, and the InfiniBand QoS mechanism (virtual lanes) in Section 3.3.2 on page 35.

Our discussion is based on the structure and the components of an InfiniBand network, see Figure 3.4 on page 33 for an example. An InfiniBand network may be constructed from switches, host channel adapters (HCA) in computer systems, and routers, connected by point-to-point links, respectively. InfiniBand defines a second type of channel adapters that connect I/O devices to the network, however, we do not mention them in the subsequent discussion, because they can be considered as a special type of cluster node, if needed.

Networks (or fragments thereof) that only consist of switches and channel adapters are called subnets in the InfiniBand architecture. This definition is relevant, because subnets form an administrative domain for network management. Several subnets may be connected by routers to form larger networks. In the following discussion, we assume an InfiniBand network comprising several switches in a non-trivial topology, which potentially results in non-uniform packet delays between different hosts (e.g., between hosts connected to the same switch, compared to hosts connected to different switches). We exclude the minimal case of two HCAs connected directly without a switch.

We based the InfiniBand-specific parts of our analysis on the *InfiniBand Architecture Specification, Volume 1, Release 1.2.1*, see [38]. We refer the reader to that document for more in-depth

details. However, the book [80] serves better as an introductory reading, even though it is not based on the recent revision of the InfiniBand Architecture Specification.

InfiniBand Addressing

InfiniBand access in a virtualized environment poses a set of new challenges compared to LAN technologies such as Ethernet. One of them is the InfiniBand architecture's addressing scheme, which is strictly associated with the physical topology.

InfiniBand differentiates between two types of node addresses: within a subnet (i.e., not passing routers), switches forward packets based on local ids (LID), which are unique only within a subnet, whereas routers exchange traffic between subnets via unique global ids (GID). Each port of a HCA or router is assigned both a local and a global id. Routers translate GIDs to the LID of either the next router, or the destination host, before handing a packet over to a switch for transport in or through a subnet.

Within a node, traffic is assigned to a distinct communication endpoint, a queue pair, using a numeric identifier (a queue pair number, QPN). Running several VMs on one node adds another level of packet routing: Packets that reach an InfiniBand HCA in a certain host have to be delivered to their destination OS instance (one of the guests or the host).

InfiniBand allows to assign several LIDs to one port of a HCA, originally intended for multi-pathing (defining different paths between two hosts through the network topology and allowing software to select which one to use). This feature, called LID Mask Control (LMC), is enabled by setting a parameter with the same name in HCAs. It specifies, how many of the low order bits of the destination LID of received packets are to be ignored (masked), and causes a HCA to accept packets addressed to LIDs in the interval $[\mathit{base\ LID}, \mathit{base\ LID} + 2^{LMC}]$ (the LMC least significant bits of the base LID have to be zero).

InfiniBand Addressing in Virtualized Environments

Without virtualization, a HCA is used by exactly one OS and therefore LIDs discern between OS instances. When a HCA is used in a virtualized environment, the virtualization concept determines, at which addressing level you can tell different VMs apart. We have found two alternatives in existing virtualization approaches:

- Dedication of an InfiniBand HCA to a single VM with PCI passthrough: The assigned LID is only used by this single VM, thus there is a clear correlation between LIDs and VMs. Every mechanism that serves to isolate non-virtualized nodes (based on differentiating HCAs and LIDs) will work in this setup, too.
- Sharing a HCA using paravirtualization or SR-IOV: The LID assigned to the HCA is shared between all VMs and the host OS. Queue pair numbers are allocated by the host OS. Other nodes cannot easily determine to which OS a communication endpoint, a queue pair addressed by a (LID,QPN)-pair, belongs. Two such pairs with the same LID can address the same or different virtual machines, which can further belong to the same or different virtual clusters. Clearly, isolation requires great care with this option, because isolation schemes that differentiate traffic only based on the destination/origin LID fail

to distinguish individual VMs. Further, isolation schemes that work at the granularity of HCAs fail, too, because they cannot differentiate and isolate between the VMs running on one host, and thereby using the same HCA.

InfiniBand Isolation Mechanisms

The InfiniBand transport layer protocol provides a partitioning scheme to form groups of nodes and isolate them from each other. Management tools (a subnet manager) configure each node to be a member of one or several partitions. Each packet carries an identifier of the partition it belongs to (a partition key, P_Key), which has a length of 16 bits. Each node contains a table with the identifiers of partitions it is a member of. When a node is member of several partitions, an application can decide which partition to use for a connection or datagram by specifying an index into this table. The most significant bit of a partition identifier (the 16-bit P_Key) indicates the type of membership[12] in a partition, so 32768 different partitions can be used.

An InfiniBand HCA must check for each received packet that it belongs to a partition the HCA is a member of (the packet's P_Key is listed in the HCA's membership table), and otherwise drop the packet. Switches optionally support a mechanism called partition enforcement, which allows to filter packets based on partitions in the switch. It is configured separately per port using a table of partition identifiers as in HCAs. Using this feature, we do not have to rely on the partition filter (and its integrity) in a HCA alone.

In a nutshell, InfiniBand allows to form groups of nodes (up to 32768 groups with an unrestricted number of members), whose members can only communicate within each group. Network traffic and nodes are assigned to a group (partition) using identifiers in each packet and in a membership table in each HCA. Although HCAs have to drop packets from foreign groups, switches provide an additional filter mechanism. We shall employ this isolation mechanism to restrict virtual nodes of a virtual cluster to communication only within the virtual cluster, by assigning all virtual nodes of a virtual cluster to the same isolation group (partition).

InfiniBand Quality of Service Mechanisms

The InfiniBand architecture defines a flexible QoS mechanism, which allows to separate network flows and to schedule network bandwidth in a very flexible and configurable way. We shall employ this mechanism to allocate shares of network bandwidth to virtual clusters. In this section, we present and introduction to the QoS mechanism, its nomenclature, and a brief description of its mode of operation.

InfiniBand allows to differentiate network flows into 16 service levels (SL) and offers up to 16 send queues per port, called virtual lanes (VL), to arbitrate physical link bandwidth in a flexible way. The InfiniBand architecture does not specify QoS policies for user data by itself,

[12] InfiniBand Partitioning differentiates between full members and limited members of partitions. Two limited members of a partition cannot communicate with each other, but all other combinations can. This differentiation allows to further refine the allowed communication relationships than partitions alone.

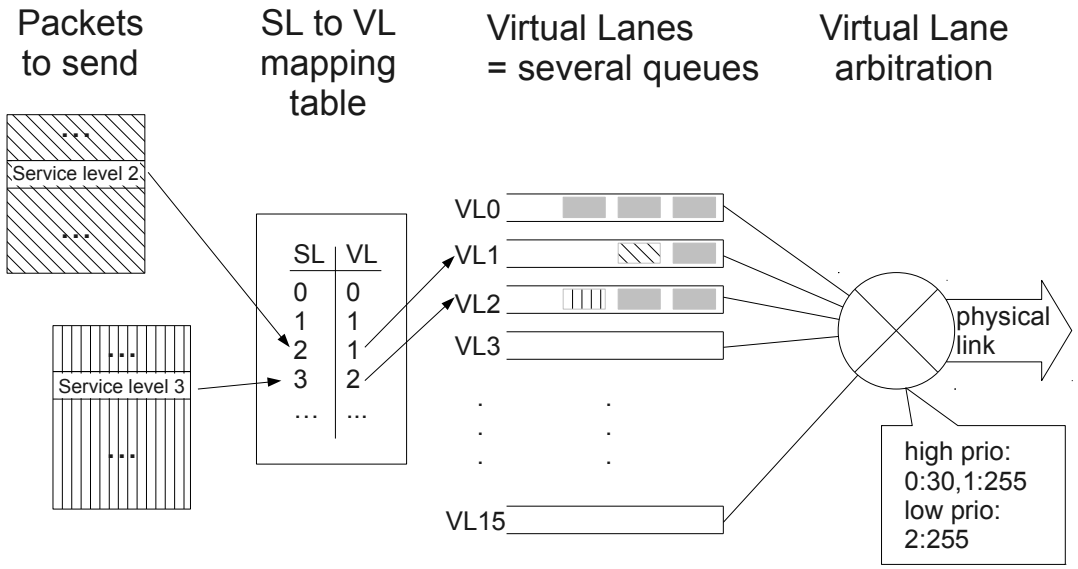

Figure 3.5: The InfiniBand QoS mechanism: several packet send queues (virtual lanes), service levels for traffic differentiation, and the mechanisms SL to VL mapping and VL arbitration. The examples shows four different SLs, which are mapped to three VLs. VL arbitration prefers VLs 0 and 1, with a higher weight for VL 1. Packets from VL 2 are only sent, when there are no packets waiting for VL 0 or VL 1.

but defines two flexible mechanisms: each packet traverses the service level to virtual lane mapping, and the virtual lane arbitration, before it is sent on a physical link. Each packet is handled according to the SL field it carries in its header[13], see Figure 3.5 on page 36.

The first step is the SL to VL mapping: Each packet is sorted into a VL based on its SL header field and a SL to VL mapping table. HCAs and routers contain one table for each port. Switches select a mapping table based on the inbound and outbound ports of a packet and therefore contain one such table for each possible combination of inbound and outbound ports. Packets can be filtered: Virtual lane 15 is reserved for management packets[14] and specifying VL 15 in the mapping table causes packets of the respective SL to be dropped. Compare Section 7.6.6, p. 190 of [38] for a description of this filter method and references to more details.

The second step is VL arbitration: Packets are chosen from VLs and sent onto the attached physical link based on two arbitration tables, a low-priority and a high-priority table. Each table entry designates a VL and a transfer credit (in multiples of 64 bytes). VL arbitration iterates over the table entries, sends packets from the VL indicated by an entry until the accumulated

[13] The SL is contained in the Local Routing Header, which is present in all InfiniBand packets.

[14] The one QoS policy that the InfiniBand architecture does define, is that management packets are sent before all other packets. They are sorted into VL 15, which cannot be used by user packets.

lengths of sent packets exceed the entry's transfer credit, then continues to the next entry. The high-priority table is walked as long as there are more packets waiting in the VLs it refers. Otherwise, VL arbitration iterates over the low-priority table. See [38] for further details.

In summary, we can employ the InfiniBand QoS mechanisms (service levels, virtual lanes, SL to VL mapping, and VL arbitration) to implement bandwidth shares. Traffic is associated with bandwidth shares using the service level identifier in every packet. In our case, each virtual cluster is assigned a distinct service level and thereby a share of network resources. Each service level is mapped 1:1 to a virtual lane, so virtual lane arbitration effectively schedules network bandwidth to traffic of virtual clusters.

InfiniBand Management Interfaces

The InfiniBand architecture defines management interfaces and protocols that are used to configure an InfiniBand network (a subnet) and to setup, for example, isolation and a QoS scheme, using the mechanisms described above (partitions and virtual lanes). These protocols utilize a special type of network packet, called management datagrams (MAD).

There are two classes of implementations for each protocol: managers, and management agents. One or more managers employ the management protocol to query information about nodes or to query or modify configuration settings of switches and HCAs. For this purpose, they exchange MADs with the management agents in nodes, which act upon the manager's requests.

All InfiniBand devices (e.g., HCAs, switches, and routers) have to support a basic set of management protocols and therefore implement a management agent for these protocols. The most important one is the subnet management protocol, which a subnet manager uses to assign addresses (LIDs and GIDs), to fill a HCAs partition membership tables, and to configure the QoS mechanism in every port (virtual lanes), amongst others.

As a consequence, we can employ all the advanced features of InfiniBand using a standardized interface, independent from the hardware actually used. In contrast, advanced mechanisms of Ethernet, such as VLAN, have to be configured with vendor-specific protocols.

3.3.3 Transformation to Virtualized Environments

The InfiniBand mechanisms for isolation and QoS, as they are designed, work at the granularity of ports (e.g., of a HCA, or a switch). These mechanisms, especially partitions for isolation, can be applied in straight-forward manner when a port is used by only one OS instance. However, in a virtualized environment several OS instances share a port: All the VMs running on a host access the InfiniBand interconnect via the port of the HCA built into the host. In this setting, the granularity of ports is clearly not sufficient to separate individual OS instances. We have to reconsider how we can implement and enforce partitioning and QoS policies in the presence of virtualization.

After discussing where the existing mechanisms fail, we propose extensions that allow to use isolation and enforce a QoS scheme in virtualized environments. Subsequently, we show that we can use a similar approach to virtualize node addresses, which we employ to provide a

virtual cluster with the view of a virtual InfiniBand network (compare Section 3.3.1 on page 31).

We differentiate two cases:

1. A host runs only VMs belonging to the same virtual cluster. From the perspective of isolation and QoS, the HCA (and its port) are dedicated to one entity, the virtual cluster. The mechanisms work as in the non-virtualized case.

2. A host runs VMs belonging to several different virtual clusters. Isolation and filtering in the network has to be loosened up. A switch must pass packets of several partitions to/from a host, but cannot discern legitimate from illegitimate use of partitions by VMs on a host. Therefore, host OS and HCA have participate in enforcing network isolation.

We will focus on the second case in the following discussion. Contrary to Section 3.3.2 on page 33, where we introduced the InfiniBand mechanisms for isolation and QoS, we discuss the virtualization of the InfiniBand management interface first (in Section 3.3.3 on page 38), because it forms the foundation for enforcing isolation and QoS in virtualized environments. We shall discuss these issues subsequently in Sections 3.3.3 on page 40 and Section 3.3.3 on page 41. Then, we describe how we virtualize node addresses in Section 3.3.3 on page 42, which forms the second foundation for providing virtual clusters with a virtual network view, besides virtualizing and restricting access to the InfiniBand management interfaces.

The network virtualization we describe for VMs can be realized as well in an OS to provide separate virtual network views and network isolation on the level of processes. However, we employ virtualization to provide complete VMs as virtual cluster nodes and therefore provide InfiniBand network isolation in the SR-IOV driver backend in the host OS, completely transparent to the guest OS.

All the extensions we propose require changes only to the drivers running in the host OS. As a result, we do not rely on the integrity of drivers running in the guest, and, more importantly, a user can simply use existing drivers installed per default and does not need to modify his OS environment, libraries, or his own code base.

Intercepting Access to Management Interfaces

The InfiniBand isolation and QoS mechanisms are all configured via the InfiniBand subnet management interface — that is, via sending appropriate packets (MADs) to switches and HCAs. Access to this interface would allow to override network isolation and to modify bandwidth shares. Clearly, we must prohibit users to apply such modifications to the physical network, and therefore restrict their access to the management interfaces of the physical network. On the other hand, we want to enable a user to configure his share of the network, in a controlled way (i.e., without a breach of isolation), via the standard InfiniBand management interfaces, so that he can use regular tools for this purpose. In this section, we show that management interfaces have to be put under control of the host OS anyway, and thus we can intercept a VM's management packets and redirect them to a topology simulator that implements a virtual view of the InfiniBand network for management tools. Further, we introduce a second mechanism,

provided by the InfiniBand architecture, that protects the configuration of the physical network in case a user breaks out of a VM and gains full control of a HCA.

The network packets employed in the InfiniBand management interfaces (MADs) must be sent from and addressed to special communication endpoints (queue pairs) in a HCA, namely the queue pairs numbered 0 (subnet management protocol), and 1 (all other management protocols, they allow the OS to mark further queue pairs as privileged); compare Section 3.9 in [38]. The InfiniBand architecture demands, that access to these special communication endpoints is restricted to privileged users (e.g., an OS kernel), compare Section 10.2.4.5 in [38], which clearly must be fulfilled in a virtualized environment, too.

In the InfiniBand stack of a Linux running natively, a driver in the kernel has direct access to the special queue pairs, and user level management tools must pass MADs to the kernel. The approach taken in the SR-IOV implementation on Mellanox ConnectX-2 HCAs keeps access to the special queue pairs restricted to the host OS kernel, see Figure 3.6 on page 40. From the perspective of management applications, nothing has changed: They continue to use their OS's MAD interface to send or receive management datagrams, whether they run on the host or guest OS. The behavior of the guest OS InfiniBand driver differs: Instead of passing them to the special queue pairs directly, the guest OS hands over these requests to the host OS via proxy queue pairs that tunnel management datagrams from guest to host. Eventually, the host OS employs its access to the special queue pairs to send the management datagrams on behalf of the guest management tool.

We expect other future virtualization implementations for InfiniBand to follow a similar approach and also restrict access to the special queue pairs to the host OS. The paravirtualization approach in [53] leaves out the InfiniBand management interface, so the guest OS has no access at all. To our knowledge, there are currently no other SR-IOV implementations for InfiniBand HCAs besides that on ConnectX-2 HCAs by Mellanox, which we employ in this work.

Neither guest user-level applications, nor the guest OS kernel can send management datagrams to the network completely on their own. They have to pass such datagrams to the host OS (indirectly via the HCA), which sends them on the behalf of the guest. Here, we place our interception mechanism. We modify the part of the host OS driver that processes MAD send requests by guests and replace the forwarding to the HCA with a redirection to a user-level tool running in the host OS.

As a result, we deny VMs access to the management interface of the physical network, and at the same time enable a flexible virtualization scheme of the InfiniBand management interfaces. From the perspective of the guest, there is no difference between a physical network and a management protocol simulator attached to the MAD redirection.

One might argue that this protection measure does not help against a user that breaks out of his VM and reaches control of a HCA via the host OS. Further, it does not protect subnet configuration against subjects with legitimate full access to a HCA in the subnet. For this purpose, every subnet management agent (the part of a node reacting to subnet configuration requests) can be configured to accept management requests only from specific managers. Each node has a 64 bit long protection key (M_key) and two protection enable flags. This protection scheme, when enabled, causes an agent to ignore any subnet management requests that do not contain the correct protection key (as a field in the MAD).

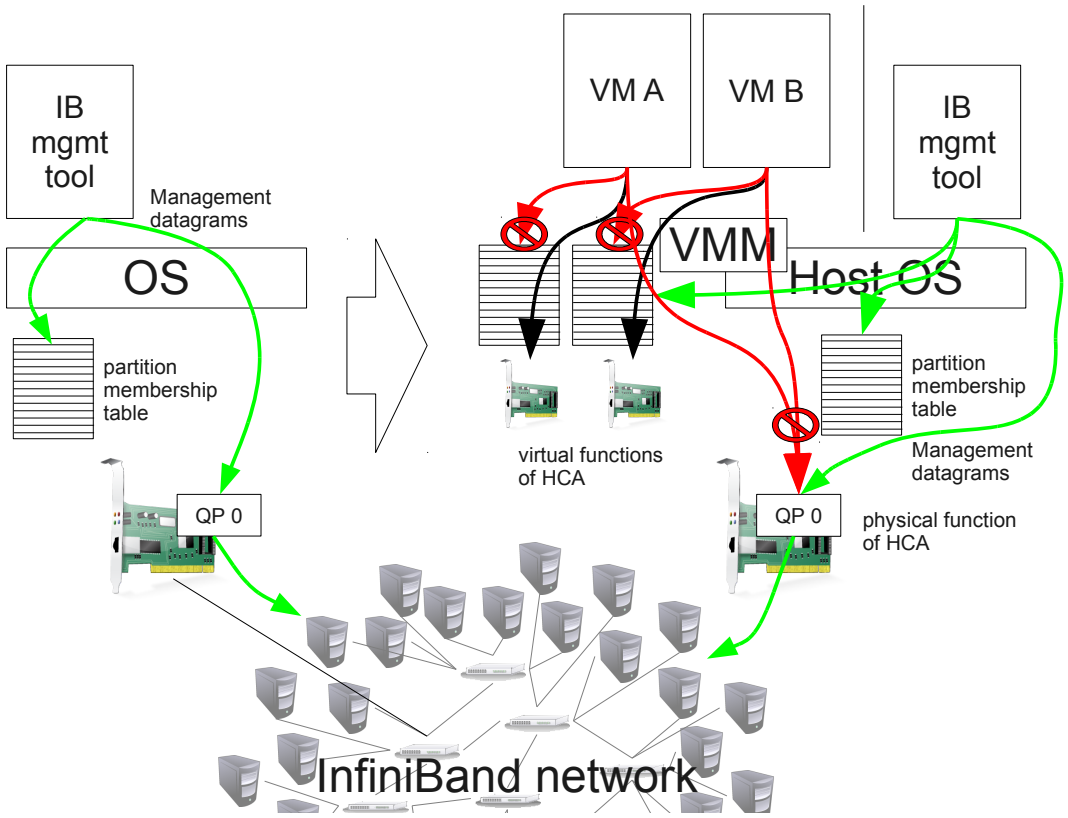

Figure 3.6: InfiniBand management interfaces and isolation mechanism with virtualization. SR-IOV restricts access to the host OS and provides a separate partition membership table for each VM.

If a subnet manager employs different random values as protection keys in all subnet management agents[15], a malicious user cannot overcome this protection mechanism by brute force. A malicious user cannot brute-force a protection key (M_key) offline, but has to send MADs to a subnet management agent. Therefore, the potential for parallelization of the attack is limited by the bandwidth of physical links. We estimate such an attack to take more than 3000 years, assuming a 4xQDR link at 40 Gbit/s, and therefore consider it infeasible.

Enforcement of Isolation

The InfiniBand architecture defines a partitioning mechanism, which allows to form groups of nodes, and restrict them to communication within their group. We have introduced this mechanism and mentioned that we employ it to isolate network traffic between virtual clusters in Section 3.3.2 on page 35. However, InfiniBand partitions were designed with the granularity

[15]The open subnet manager opensm, a widely used open-source implementation, in current versions only supports one value for all M_keys in the subnet and does not enable the protection flags. We have to extend opensm to achieve effective protection.

of (physical) network ports in mind, and virtualization and several VMs that use the same HCA and thereby network port, challenge its effectiveness.

When VMs of different virtual clusters run on a host, the switch port that the host is attached to has to permit packets from/to several partitions. So, the host OS and the HCA are responsible to constrain each VM to use only the partition it belongs to, because the switch cannot determine from which VM a packet was sent.

At the InfiniBand software interface, one cannot specify which partition identifier (the P_key) to use, but an index into the partition membership table of the HCA. So, the host OS has to ensure two properties: A user in a VM may not be able to modify the partition membership table, and a user can only refer to table entries of partitions that belong to his virtual cluster. We can achieve both with SR-IOV drivers for Mellanox ConnectX-2 HCAs. First, modifying the partition membership table is a privileged command, which only the driver in the host OS can issue. Such modifications are usually triggered by a subnet manager via the subnet management protocol. As discussed above, we can lock a VM out of the management interfaces of the physical subnet, to ensure it cannot send such requests. Second, the SR-IOV implementation virtualizes the partition membership tables. Each virtual function (the virtual PCI device interface which a VM uses) is assigned a separate table, see Figure 3.6 on page 40. So, we can clearly separate the partitions different VMs can use.

Enforcement of Bandwidth Shares

We have introduced the InfiniBand QoS mechanisms above in Section 3.3.2 on page 35. An identifier in every packet (service level) allows to sort packets into different send queues at each network port and a transmit scheduler (virtual lane arbitration) then arbitrates the physical link's bandwidth between the send queues according to a flexible configuration scheme.

Key to assigning bandwidth shares to virtual clusters is, that each virtual cluster only uses the service level(s) assigned to it. The service levels designate, which bandwidth share each network packet accounts to. So, if a malicious user can employ the service level identifier of another virtual cluster in his own network traffic, he can illegitimately use the other virtual cluster's bandwidth. As a result, we have to enforce that network traffic originating from each virtual cluster (i.e., from each of it's virtual nodes) only uses the service level assigned to this virtual cluster.

In the introduction of service levels (SL) in Section 3.3.2 on page 35, we mentioned that packets can be filtered in switches based on SLs. This filter can be configured at the granularity of ports only, and therefore virtualization poses the same challenge for SL filtering as for partitioning. A switch has to admit packets with the SLs from all VMs running on a host, so the host has to constrain a VM to the SLs it is allowed to use.

The way a SL is specified differs between InfiniBand transport types:

- For reliable connected, unreliable connected, and reliable datagram type queue pairs, the desired SL is a parameter to the commands issued to setup the queue pair. These commands are issued by the host OS on behalf of the client, so we can add a wrapper

for this command that either denies requests for SLs a VM may not use, or transparently changes the SL parameter.

- With unreliable datagram queue pairs, the SL to use in a datagram is specified in the send request of the datagram itself. Normally, a user application can post such send requests directly to the HCA via the OS-bypass mechanism. To constrain SLs, we can disable OS-bypass for unreliable datagram queue pairs[16], thereby forcing an application to send such datagrams via syscalls to the OS driver interface. These will eventually result in requests to the host OS driver to issue the send command, where we can filter requests based on SLs. Regular use of the unreliable datagram transport will suffer from increased latency, but the workload we focus does not use it: Parallel applications based on MPI commonly utilize the reliable connected transport type (see [81]).
  If the demand occurs, a SL filter in the HCA without adverse influence on performance would require an extension to the HCA hardware. We propose a bitmap for each unreliable datagram queue pair that is configured by the host OS and restricts which SLs send requests posted to this queue pair may specify.

Node Address Virtualization

So far, we have discussed how we restrict a guest from modifying the configuration of the physical network by redirecting management packets from the guest to a network virtualization layer (see Section 3.3.3 on page 38. At the same time, we made possible to provide a user the impression of a dedicated InfiniBand network that he can configure with regular management tools, like a physical network, by appropriate design of the network virtualization layer. Then, we described how we can employ the InfiniBand mechanisms for isolation and QoS safely in a virtualized environment. Finally, in this section, we complete the building blocks for the virtual network view we provide to a virtual cluster, see Section 3.3.1 on page 31, by introducing the virtualization and translation of node addresses.

Actual network traffic and packet forwarding in switches uses the LIDs that are assigned to HCA ports by a subnet manager. Therefore, communication or configuration requests to the physical network infrastructure have to use such physical LIDs (e.g., for requesting a HCA to open a connection to a remote node). We can, however, enable VMs and the applications therein to use other node addresses, which do not have to correlate with those physically used, thereby providing each virtual cluster its own node address space.

Virtualizing node addresses is one key requirement for the virtual view of the cluster interconnect that we want to provide for virtual clusters, as introduced in Section 3.3.1 on page 31. A user can employ existing InfiniBand management tools to configure his share of the network just like a physical InfiniBand network, including the assignment of LIDs to nodes (this assignment is commonly the second step in setting up an InfiniBand network, after discovering its topology).

[16] With the ConnectX-2 HCA, OS-bypass application has to be enabled, if desired, when a queue pair is created, a privileged command issued by the host OS.

We cannot use these user-assigned LIDs on the physical network, because LIDs must be unique in a subnet and the assignments of different virtual clusters would typically collide (it is common to assign consecutive LIDs, starting with *1*). If we considered the user-assigned LIDs only in the virtualization of the InfiniBand management interface, and employed the physical LIDs in the interfaces for actual use of the interconnect, the user would perceive an inconsistence: a user's applications would use one address, and his management tools would use a different one, for the same virtual node.

We solve this problem by virtualizing LIDs also in the InfiniBand communication interface and thereby provide a consistent, virtual node address space to a virtual cluster, for applications as well as for management tools.

So far, we discussed how we constrain VMs that use an HCA virtualized with SR-IOV by adding checks and wrappers to the host OS driver, so that we guarantee isolation and legitimate use of bandwidth shares in a virtualization environment. We employ the same approach to virtualize node addresses (LIDs) and thereby implement virtual node address spaces for VMs.

The applications and drivers inside VMs exchange LIDs with the virtualized interface of the HCA at four occasions:

1. Establishing communication associations for the reliable connected, unreliable connected, and reliable datagram transport types.
2. Issuing send requests to the unreliable datagram transport protocol.
3. Parsing completed receive requests (i.e., filled receive buffers) from the unreliable datagram transport protocol.
4. Querying the node address of to the HCA they use (required as addressing information to establish connections with peers).

For the reliable connected, unreliable connected, and reliable datagram transport types, the host OS can step in while a communication association is initiated (this is a command the host OS issues to the HCA), without further influence on the performance of actual communication operations (OS-bypass as usual) and translate the destination node address. The same applies to the case of intercepting connection setup requests to prohibit illegitimate use of bandwidth shares, discussed in Section 3.3.3 on page 41. After a connection to a remote node has been setup using the reliable connected, unreliable connected, or reliable datagram transport type, node addresses are not used at the software interface any more, but only by the involved HCAs internally and on network.

For the unreliable datagram transport, we have to completely virtualize queue pairs to effectively alter LIDs, or add a mechanism to the hardware. With unreliable datagrams, the destination of each datagram is specified in the send request of that datagram and the origin of received datagrams is stored in the receive completion queue entry (an entry in the receive queue, whose buffer has been filled with received data) by the HCA. The queues containing these requests are shared between an application and the HCA, and therefore have to contain physical LIDs, which the guest application does not know. With virtualized queue pairs, the host OS intercepts each sent and received unreliable datagram and can translate the destination and origin LIDs from the virtual cluster's virtual node address space to the node addresses used on the

physical network. Virtualizing queue pairs causes overhead significant overhead compared to OS-bypass, but we expect that hardly any applications use unreliable datagrams[17].

In setting up connections or sending datagrams via the unreliable datagram transport, a user application specifies an LID for the HCA to use. In contrast, an application may query a HCA for the assigned LID. This address is needed to exchange InfiniBand connection information with a peer prior to establishing an InfiniBand connection (as done by OpenMPI). For the purpose of node virtualization, we have to intercept this kind of requests, too, and return the LID assigned to the node in the virtual cluster's view of the network. Querying the HCA for the assigned address is a privileged operation that has to be issued by the host OS. A driver running in a VM has to ask the host OS for this information, so we modify the host OS driver to reply to such requests with a virtual LID.

We propose to use a direct mapping table, indexed by virtual destination LID, to map from a virtual cluster's virtual LID space to the physical LID space. Inversely, we store the LID to be reported to the VM as assigned to the HCA port as a field in the VM's metadata. The memory overhead of the translation table is 128 KB per VM (2^{16} possible LIDs, each mapping to a 16 bit LID) and the compute overhead of the LID translation is negligible, compared to the complexity of the overall action: leaving guest mode, several I/O operations to post the command to the HCA, returning to guest mode, only to name a few operations. In addition, for the common transport modes, LID translation only happens when establishing a communication relation, not during actual communication operations.

3.3.4 Mapping Virtual Network Views to the Physical Network

So far, we have introduced the virtual network view we want to present to each virtual cluster and the mechanisms InfiniBand provides for isolation and QoS. In addition, we discussed how we adapt these mechanisms to virtualized environments. In this section, we combine the provided InfiniBand mechanisms and further virtualization mechanisms, to implement the virtual network view for virtual clusters. We present the basic configuration of the physical InfiniBand network first. Then, we discuss how we merge and map the network configurations of all virtual clusters to the configuration of the physical network and which basic policy we follow in the process.

The physical InfiniBand network is controlled via the InfiniBand management interface by a subnet manager of the cloud provider. No one else can access the management interfaces of the physical network. See Section 3.3.3 on page 38 for a discussion on how we achieve this, and Section 4.3.1 on page 76 for an evaluation of the key-based protection mechanism on actual hardware.

The cloud provider's subnet manager assigns node addresses in the physical network (LIDs) — these node addresses are used by packet forwarding. Every node is assigned several consecutive node addresses using the LMC mechanism. Traffic of cloud management always uses the first out of the range of assigned addresses (the base LID). We use the additional addresses to

[17] A main application for the unreliable datagram transport types are management datagrams. However, they have to be sent from QP0 or QP1, and use of these queue pairs is restricted the host OS, anyway.

configure custom routing for virtual clusters (in fact, a type of the multipathing that the LMC mechanism is designed for).

We map node addresses, custom routing, custom isolation setup, and custom QoS configuration from each virtual cluster's virtual network view to the physical network. The mapping process differs, depending on whether a virtual cluster is in basic or full network virtualization mode (compare the introduction of the virtual network view in Section 3.3.1 on page 31). Our goal of providing network and performance isolation has precedence over mapping all details of a virtual cluster's configuration to the physical network. When configuration resources of the physical InfiniBand network (e.g., partitions or virtual lanes) become scarce, they are utilized first for network and performance isolation between virtual clusters and only second for fine-grained configuration within a virtual cluster.

Node Addresses

A virtual cluster in basic mode (i.e., without a subnet manager started by the user) uses the first of the physical node addresses, as assigned by the cloud provider's subnet manager. The physical node addresses are directly visible within the virtual cluster and there is no translation of node addresses. When a virtual cluster is switched to full virtualization mode, node address translation is switched on for each VM of the virtual cluster and each VM is dedicated one of the additional node addresses. That way, management and user applications in a virtual cluster both consistently use their own node address assignment.

Custom Routing

Virtual clusters in basic mode use the routing configured by the cloud provider. Virtual clusters in full virtualization mode can use a custom routing scheme, because each virtual node of the cluster is assigned a dedicated physical node address (out of the range of addresses assigned to its host) and packet forwarding in switches is based on node addresses. For each virtual cluster, the mapping process extracts the routing tables of each switch from the topology state machine that handles the virtual cluster's management packets and therefore reflects the configuration intended by the user of the virtual cluster. The routing tables are translated from referring to custom node addresses to using the node addresses used in the physical network, before they are incorporated into the routing tables of the physical network. That way, a user can configure a custom routing scheme inside his virtual cluster, which is actually used for the user's traffic in the physical network.

Recursive Network Isolation

The physical subnet manager always configures a basic network isolation scheme that associates each virtual cluster a separate partition. In addition, traffic for cloud management and non-HPC traffic each are assigned a separate partition, too. Although virtual clusters in basic mode use the same node address as the host, they are still isolated from other virtual clusters and from cloud management traffic, because InfiniBand network isolation is based on partition identifiers in each packet, not on node addresses, and the tables containing the partition identifiers of a (virtual) node are separate per VM.

A user of a virtual cluster can configure isolation by himself to further partition his virtual cluster. We map such recursive partitions to the flat space of physical partitions. For this purpose, we extract partition membership tables from a virtual cluster's virtual topology, translate the partition identifiers if required to avoid duplicate uses in the physical network, and store them into the partition membership tables of the respective VM. The available partition identifiers will only become scarce in very large physical clusters with very fine-grained isolation (there are 32768 identifiers). The translation of partition identifiers is inherently transparent, because the InfiniBand software interface uses indexes to the partition membership table instead of actual identifiers.

Recursive QoS Policies

In contrast to partitions, the resources to implement QoS policies with InfiniBand (service levels and virtual lanes) are much more restricted in InfiniBand. On each port, traffic can be separated in up to 15 traffic classes (based on the service level identifier in each packet) and scheduled separately for transmission to the physical link. These 15 traffic classes must suffice to separate traffic of virtual clusters and to implement the QoS policies defined in a virtual clusters — each user could himself try to use up to 15 traffic classes in his configuration.

In merging and mapping the virtual clusters' QoS configuration to the physical network, we first determine which virtual clusters use each link (derived from the routing configuration). We then assign a service level to each virtual cluster, so that all virtual clusters contending for a link use a different service level (virtual clusters that do not share a link can be assigned the same service level). We then apply the network priorities of the virtual clusters (e.g., all have equal bandwidth shares, or one cluster is prioritized) and generate virtual lane configurations for each link. That way, network performance isolation is achieved first.

If there are some service levels out of the 15 still available, we can assign additional service levels to a virtual cluster. The service levels used in the configuration inside a virtual cluster then get compressed to the service levels available for the virtual cluster. The traffic scheduling policy configured by a user is applied to divide the bandwidth share assigned to the virtual cluster. Clearly, it makes no sense to assign additional service levels to a virtual cluster without a custom QoS setup (such as virtual clusters that do not run their own InfiniBand management tools).

3.4 HPC Cloud Management

In the previous sections of our approach, we have discussed how we turn physical hosts into virtualized environments for virtual HPC nodes and how we virtualize the physical cluster network. All the mechanisms we introduced and the policies we discussed have to be configured and controlled to automatically provision elastic virtual clusters and assign them a share of the physical cluster network. These are the tasks of HPC cloud management. There are cloud management frameworks that provide users with elastic virtual clusters. Our contribution is to extend these frameworks with the automatic management of cluster networks, such as InfiniBand. We aim for an automatic setup of network isolation and bandwidth shares for

virtual clusters. We add the cluster network's topology as a new abstraction to cloud management, which is incorporated in VM placement policies. In addition, we extend the interaction between the cluster job scheduler inside a virtual cluster and HPC cloud management to incorporate topology information. See Section 4.1.4 on page 56 and Section 4.4.1 on page 78 for a description and an evaluation of our prototype that automatically configures InfiniBand partitions for network isolation between virtual clusters.

In this section, we first provide an analysis of the restrictions that the use of a cluster network poses on cloud management, and what functionality existing cloud management frameworks already provide towards managing an HPC cloud with a distinct cluster network (Section 3.4.1 on page 47). Then, we briefly discuss the required extensions that enable a cloud framework to manage a cluster network and how we employ a tight cooperation between cluster job schedulers and cloud management to overcome the restrictions (Section 3.4.2 on page 49).

3.4.1 Analysis

In this section, we present the constraints and restrictions the use of a distinct cluster interconnect, such as InfiniBand, poses on cloud management: Connection state in HCAs currently makes transparent live migration impossible and the influence of the physical network topology on performance and restrictions for QoS configuration pose constraints on the placement of VMs in the physical cluster. In addition, we discuss the features existing cloud management frameworks provide for HPC cloud management.

Virtual Machine Live Migration and InfiniBand

Live migration of VMs [19] enables the transfer of VMs between hosts at runtime, with only minimal downtime. In the ideal case, the migration is transparent to the migrated VM as well as to all peers that communicate with the migrated VM. Live migration of VMs is typically used for load balancing and to move VMs away before a host is shutdown for maintenance or because of errors (e.g., when a failure of the cooling system is detected).

There is currently no known way to migrate VMs with access to InfiniBand in a way transparent to user-level applications that maintain active connections over InfiniBand, because the state of open connections and the mode address of a VM can not be migrated with the VM. We shall discuss the reasons, why migration with InfiniBand is hard, compare the situation to Ethernet, and present a solution for migration that involves user-level applications.

With current InfiniBand HCAs, the protocol state of open connections is part of the internal state of an HCA. Thus, it cannot simply be transferred with a VM's memory image during migration. In addition, when several VMs share an InfiniBand adapter and thus a node address, this address cannot be migrated away with one VM, because of the other VMs still using it (see Section 3.3.2 on page 34). Consequently, the connection peers of a VM being migrated would have to be notified of the address change, and consequently modify their connection state. Because node addresses are also used at the user-space interface of InfiniBand, user-level software has to be modified to handle migrations that change a VM's node address.

In contrast, transparent live migration works with TCP/IP over Ethernet, because connection state is kept in the guest OS's protocol stack and thereby is migrated with the memory image of the VM, and addressing with IP on Ethernet is much more flexible. Usually, each VM has a distinct MAC address, which can be migrated with the VM, whereupon packet routing automatically adapts to the new location in the network topology. Even when a VM must use a different MAC address after migration, the indirection of using IP addresses in applications and MAC addresses in the link layer allows to hide this change from user-space.

Live migration of VMs with InfiniBand connections is possible, when the migration process involves user-level applications. Huang has proposed the nomad framework in his PhD thesis [36], an extension to the InfiniBand paravirtualization approach in [53] to accomplish live migration of VMs. The nomad framework is based on the fact that most HPC applications do not use InfiniBand directly, but employ a communication library such as MPI. By modifying the MPI implementation, the approach conceals migration from HPC applications, although the MPI library they use is involved in migration. Open InfiniBand connections are closed before, and re-established after migration with cooperation by connection peers, which are notified before and after the migration, and cease communication with the migrated VM in the meantime. As a result, connection state does not have to be transferred, and node addresses can be changed freely in the course of migration.

We aim at transparent operation without the need to change user software inside virtual cluster nodes, so we cannot use the Nomad approach. As a consequence, we currently cannot use live migration for VMs with InfiniBand access. Once started, a VM is bound to the host it runs on. Live migration in InfiniBand, potentially made possible by appropriate extensions to InfiniBand hardware, is a worthwhile topic for future work.

Constraints on VM Placement

In contrast to general-purpose clouds, our architecture for a cloud environment for HPC workloads provides performance guarantees for the cluster network, because HPC applications require bound latencies. Since cluster network performance, as experienced by a virtual cluster, depends on the placement of nodes in the physical topology, performance guarantees clearly pose constraints on the placement of VMs in an HPC cloud. In this section, we discuss some of these constraints.

General-purpose clouds typically employ Ethernet networks and share these networks in a best effort manner. Each individual network link is shared by an undetermined number of users. As a result, general-purpose cloud providers cannot and do not give performance guarantees for network performance, see [93] for a study on the network performance in Amazon EC2, a commercial general-purpose compute cloud. For HPC applications however, low latencies are essential for performance. Latency is increased by each hop packets have to traverse to reach other cluster nodes and by contention from other virtual clusters using the same link. So, HPC cloud management must strive (1) to place the virtual nodes of a virtual cluster as close to each other in the network topology as possible, and (2) to arrange virtual clusters in the physical cluster as separate as possible, so that network links are shared between as few virtual clusters as possible.

Since we aim for performance isolation between virtual clusters, the InfiniBand QoS mechanisms pose a limit on the number of virtual clusters that may share a link: the InfiniBand architecture defines a maximum of 15 service levels and virtual lanes, so we can separate a maximum of 15 different network flows on each link, and therefore achieve performance isolation only between a maximum of 15 virtual clusters on each link. Implementations are free to support less than 15 virtual lanes, reducing this number even more (the hardware in our prototypic setup supports 8 virtual lanes only).

Network traffic in a virtual cluster will typically flow over just a subset of the links and switches of the physical network. This subset is defined by the hosts that run the cluster's virtual nodes, the physical topology of the network, and the routing algorithm used. Consequently, this subset will differ between virtual clusters and the virtual clusters that compete for network bandwidth can be different per physical link and switch.

There is currently no way to live migrate VMs with InfiniBand transparently, as we discussed in Section 3.4.1 on page 47. So, the placement of a VM with InfiniBand access must be considered final. Cloud Management cannot reorder VMs to place the nodes of a virtual cluster closer to each other, or to reduce the number of virtual clusters that share a link.

Cloud Management Frameworks

There are several cloud management frameworks that have their origin in the research community. In this section, we briefly present two of them, *OpenNebula* [65, 84] and the *nimbus* toolkit [47, 70]. We ran our early experiments with OpenNebula, because of existing experience. Our prototype targets OpenNebula, too. Nimbus is outstanding however, because of its built-in support for virtual clusters.

Both OpenNebula and nimbus manage VMs, hosts, and disk images as abstractions, amongst others. They have a management core that delegates the actual handling of these entities to other utilities or a set of scripts. Both frameworks employ the *libvirt* toolkit [51] to start, stop, and supervise VMs, and thereby support both the KVM [49] and the Xen [12] hypervisors (because the *libvirt* toolkit supports them). Also, both frameworks support basic network isolation, which is restricted to Ethernet networks, however.

In contrast to OpenNebula, the nimbus toolkit has built-in support for virtual clusters [47]. It uses a flexible scheme for *contextualization* of VMs — that is, providing VMs with individual parameters and context information (such as DNS servers or cluster head nodes), to start VMs with different roles in a cluster (such as head node or compute node) from a single disk image file.

There is an extension that adds support for virtual clusters to OpenNebula [4]. OpenNebula also supports contextualization, although in a less flexible way than nimbus.

3.4.2 Transformation for Cluster Networks

In the previous sections, we have discussed the constraints that the use of a distinct cluster interconnect, such as InfiniBand, incurs on VM placement. Transparent live migration is not

possible when VMs use InfiniBand, and performance isolation cannot be guaranteed when more than 15 virtual clusters share a physical link. We further introduced two cloud management frameworks we employ to delegate basic cloud management. The *nimbus* toolkit even provides built-in support for virtual clusters [47]. We extend the functionality of existing frameworks with the automatic management of cluster networks to setup network isolation and bandwidth shares for virtual clusters.

We add the network topology of cluster interconnects as a new abstraction to cloud management. Cloud management maintains an abstract representation of the physical network's topology, consisting of switches, hosts, ports, and links. This representation is combined from topology discovery on the one hand (in contrast to Ethernet, InfiniBand supports this operation), and querying each host for its network ports on the other hand. Thus, the topology representation contains information about the physical hosts (the classic cloud management data) together with the network topology they are connected to (the new information we add). That way, we connect the domains of cloud management with cluster network management and enable to refer between the two domains.

The topology representation provides information about the distance of physical hosts in the cluster network to extended VM placement policies and tracks, which virtual clusters share each network link. It serves as the basic data structure for network management in the cloud management framework.

Further, we extend the interaction between HPC cloud management and cluster job schedulers running in virtual clusters to incorporate topology information. Thereby, we improve cluster resize decisions and provide a way to overcome the restriction, that VMs with InfiniBand access cannot be migrated transparently.

Elastic virtual clusters are resized dynamically by an appropriate cluster job scheduler based on demand and job load (introduced by the Elastic Site approach [56]). Information about the cluster topology can improve resizing decisions: When a cluster job scheduler decides to remove idle nodes from a virtual cluster, cloud management can suggest the nodes most distant in the topology from other node of the virtual cluster.

The other way round, a cluster job scheduler can inform cloud management about idle nodes that currently do not use InfiniBand. Such nodes can be migrated by temporarily disabling their InfiniBand access during migration, because they have no open InfiniBand network connections. So, idle VMs can be repositioned to move the nodes of a virtual cluster closer to each other in the network topology and thereby to reduce the network contention caused by the virtual clusters that use the same network link. The passed-through PCI device that provided the VM with InfiniBand access on the origin host is withdrawn before migration. After migration, the VM is assigned the device that provides InfiniBand connection on the destination host, again via PCI passthrough. The guest OS must support PCI hot plugging (Linux does) and begins to use the new PCI device with the regular initialization procedure. No connections are lost, because only idle nodes are selected for such an interruptive migration.

In this section on HPC cloud management, we have introduced the functionality we add to existing cloud management frameworks. We have analyzed the constraints a cluster interconnect incurs on VM placement and live migration and presented the abstractions and mechanisms

existing cloud management frameworks provide to manage virtual clusters. Finally, we discussed our basic approach for combining cloud and cluster network management, a topology representation that combines information about hosts and the network topology, and the intense cooperation between cloud management and cluster job schedulers that allows to overcome the restrictions on VM live migration and thereby alleviate the VM placement constraints. To prove our concept as feasible, we have constructed an early prototype that adds topology information to OpenNebula and derives the configuration for InfiniBand network isolation from both traditional cloud management data and the new topology information. See Section 4.1.4 on page 56 for an introduction to the prototype and Section 4.4.1 on page 78 for an evaluation of its practical function.

4 Prototypical Evaluation

So far, we have introduced our architecture for high-performance computing (HPC) clouds and discussed how we employ node virtualization, InfiniBand network virtualization, and an HPC cloud management framework to construct this architecture. In this chapter, we complement our approach with the presentation of our prototypic HPC cloud and the practical evaluation of some of the building blocks of our architecture. We show these components's fitness for use towards our goal, to provide virtual InfiniBand clusters for HPC workloads. Our prototype is a first step towards a complete implementation of our architecture and serves as a basis for evaluation. We use it to gain practical experience with the components that we build our architecture from.

At first, we introduce our prototype and our experimental setup on a 4-node cluster with InfiniBand in Section 4.1 on page 53. Then, we present the specific experiments we have conducted in detail. We follow the structure of our approach (Chapter 3 on page 17) and arrange the descriptions of experiments based on whether they evaluate a facet of node virtualization (Section 4.2 on page 60), network virtualization (Section 4.3 on page 75), or HPC cloud management (Section 4.4 on page 78).

We examine the influence of various Linux kernel configurations on OS noise and application performance to determine, whether we can gain significant improvements this way, or whether we have to avoid certain settings. Regarding our policy of dedicating physical cores to logical CPUs of HPC VMs, we test its antithesis on the one hand, by assessing the influence of time-sharing on HPC applications, and check on the other hand, whether we can add low-priority non-HPC VMs without disturbing HPC application performance. We analyze the practical feasibility of protecting an InfiniBand network's configuration against unauthorized modifications, which forms an essential foundation for network isolation, and provide an appropriate correction to the *opensm* subnet manager. Finally, we verify the automatic configuration of network partitions by our prototype and examine potential, but avoidable bottlenecks in OpenNebula's scalability.

4.1 Prototype

We have constructed a prototypic setup of an HPC cloud on a physical cluster consisting of four nodes, which are connected by an InfiniBand interconnect. It spans the aspects node virtualization, network virtualization, and HPC cloud management, and additionally the actual cluster operation from the perspective of a user. This prototype served as the basis for our experiments.

In contrast to an actual HPC cloud for production use, we want to run benchmarks natively in our prototype, in addition to running them inside a virtualized environment. That way, we can

determine baseline performance in the native case and assess virtualization overhead. For this purpose, we provide an HPC cluster environment inside a virtual cluster and on the host OS.

The remainder of this section is structured as follows: First, we introduce the hardware configuration of the physical cluster that forms the basis of our prototypic HPC cloud (Section 4.1.1 on page 54). Second, we present the software stack for virtualization on each node, and describe how we grant virtual machines (VMs) access to InfiniBand host channel adapters (HCAs) (Section 4.1.2 on page 54). Third, we discuss the steps towards InfiniBand network virtualization employed in our prototype (Section 4.1.3 on page 55). Fourth, we present how we employ OpenNebula to manage our HPC cloud (Section 4.1.4 on page 56). Fifth, we complete our prototype with an actual HPC cluster environment inside a virtual cluster, so that we can assess performance from a user's perspective by running HPC benchmarks in a virtual cluster. We introduce the HPC environment we have setup inside a virtual cluster in Section 4.1.5 on page 58.

4.1.1 Test Cluster Hardware

The foundation of our prototypic HPC cloud is a 4-node cluster with an InfiniBand interconnect. In this section, we describe the hardware configuration of the nodes and the interconnect.

We have run all our experiments on a cluster consisting of 4 HP ProLiant BL-460c G6 blade servers in an HP c-class blade system. The servers are equipped with Intel Xeon E5520 quad-core CPUs with hyper-threading, clocked at 2.26 GHz, 6 GB of DDR2 DRAM main memory, and direct-attached harddisk storage with a capacity of 150 GB. The servers contain two Gigabit Ethernet adapters, which are connected to two 10 Gbit Ethernet switches integrated with the blade system. We use the Ethernet connection for remote access via SSH and for the traffic of the cloud management system.

Our InfiniBand cluster interconnect comprises Mellanox ConnectX-2 InfiniBand HCAs built into each server, and an InfiniBand switch integrated with the blade system. The switch provides up to 16 ports with a bandwidth of 20 Gbit/s (4x DDR). It is based on an InfiniScale-IV switch chipset from Mellanox. The HCAs support a bandwidth of 40 Gbit/s (4x QDR), however, the bandwidth used on the physical links is constrained to 20 Gbit/s by the switch.

4.1.2 Node Virtualization

In this section, we introduce the software stack for node virtualization that we employ on each node of our test cluster we described in the previous section. In addition, we describe the tools we utilize to influence the scheduling parameters of VMs for experiments with Linux realtime priorities, the idle scheduling policy, and CPU affinity.

On all four blade servers comprising our test cluster, we installed the Cent OS 5.6 GNU/Linux distribution to the respective local harddisk. We used this Linux installation as the host OS for our prototypic HPC cloud, and as the environment for running benchmarks natively, to measure baseline performance and to assess virtualization overhead.

We upgraded the Linux kernel provided by CentOS 5.6 (a derivative of version 2.6.18) to a more recent version, because the contained kernel-based virtual machine (KVM) did not work together with current versions of QEMU. We chose version 2.6.35, because it is the most recent kernel supported by the OpenFabrics Enterprise Distribution (OFED) version 1.5.3.1[1], a bundle of InfiniBand drivers, libraries and tools released by the OpenFabrics Alliance [64]. As a result, we were able to use the same kernel version and the same InfiniBand drivers in the host OS and in VMs, to warrant comparability between results from benchmarks run natively and virtualized.

On top of the host Linux, we employ QEMU version 0.14.0, in the version adapted to KVM [49], to provide VMs as virtual cluster nodes. We grant the VMs access to an InfiniBand HCA using PCI passthrough, because BIOS compatibility issues kept us from using SR-IOV. PCI passthrough poses the disadvantage that only a single VM can access the HCA at any time, and the host OS has to give up control of the HCA, too (compare Section 3.2.1 on page 24).

Mellanox provided us with a beta version of SR-IOV drivers for the ConnectX-2 HCAs in our test cluster, in advance of the public release of these drivers. We would have preferred to test these drivers, because they allow to grant InfiniBand access to several VMs per host, as well as to the host OS, at the same time. However, the BIOS of the servers we employ does not configure the SR-IOV capability of the HCAs. The memory-mapped I/O regions of the HCA that eventually form the virtual device interfaces accessed by VMs first have to be configured and allocated as an I/O memory region of the physical PCI device, similar to the I/O memory regions used by the host OS. The BIOS is indeed responsible for this initial configuration, but fails to do so. We tried several workarounds, yet without success. So, we reverted to PCI passthrough, which we already had proved working in an early stage of our work.

For some of our experiments, we have to modify the scheduling parameters of VMs. Since VMs provided by KVM and QEMU run as regular Linux processes, we can employ tools that influence scheduling parameters of processes for this purpose. For starting VMs, we used the OpenNebula cloud management framework (see Section 4.1.4 on page 56), which in turn employs libvirt [51], a toolkit for controlling Linux virtualization mechanisms, such as KVM. The toolkit libvirt supports CPU affinity, so we can control CPU affinity via the VM definitions of OpenNebula. In contrast, libvirt currently[2] does not support to modify the Linux scheduling parameters for VMs based on KVM and QEMU. Therefore, we used the *chrt* utility to manually set priority classes and realtime priorities for VMs after they had been started by OpenNebula and libvirt.

4.1.3 Network Virtualization

So far, we have introduced the layers that form the foundation of our HPC cloud, which are node and network hardware and node virtualization. In this section, we describe the next building block of our prototype — that is, network virtualization.

Our HPC cloud architecture aims to provide a user with a virtual view of his share of the InfiniBand network, see Section 3.3.1 on page 31. An implementation of this virtualization

[1] OFED version 1.5.3.1 has been the current release at the time of our evaluation.
[2] At the time of this work, libvirt version 0.9.1 has been the most recent release.

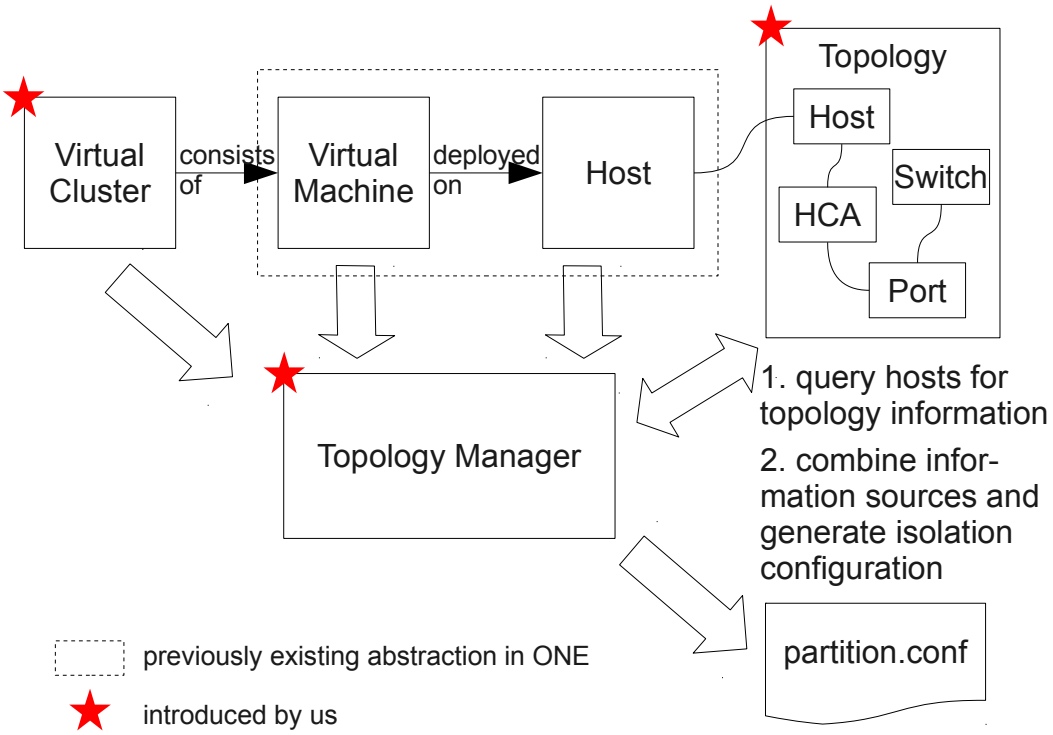

Figure 4.1: Extensions to OpenNebula that combine cluster network management and cloud management to automatically generate isolation configuration for virtual clusters.

layer is highly interdependent with an InfiniBand virtualization approach that positions the host OS between guest and InfiniBand hardware, in our case based on SR-IOV. As mentioned before, InfiniBand virtualization with SR-IOV is currently not possible in our test cluster, because of BIOS compatibility problems. Thus, we postpone a complete virtual network view to future work and focus on the elementary issues of cluster network management for HPC clouds, which are isolation (and protection thereof) and quality of service (QoS) mechanisms.

Our prototype employs the *opensm* subnet manager to configure and manage the physical InfiniBand network. It serves to implement isolation by setting up InfiniBand partitions, and QoS policies by configuring the virtual lane mechanism. Current versions of the *opensm* subnet manager do not implement the InfiniBand protection mechanism for configuration data correctly. Thus, we applied a modification to the provided *opensm* source code to correct this failure. See the evaluation in Section 4.3.1 on page 76 for a discussion of the (im)proper function of the protection mechanism and our modification.

4.1.4 HPC Cloud Management

We employ, adapt, and extend the OpenNebula cloud management framework [65, 84] for the purpose of managing node and network virtualization in our HPC cloud (see Section 3.4 on page 46). We chose OpenNebula instead of the nimbus toolkit because of existing experience.

Our extensions can be transfered to nimbus easily however, because both frameworks have a similar architecture and both employ the *libvirt* toolkit for actual host and VM management. In this section, we introduce our extensions that automatically setup InfiniBand network isolation for virtual clusters. Further, we describe how we configure PCI passthrough and CPU affinity for VMs controlled by OpenNebula, and how we employ the VM disk image management and VM contextualization mechanisms of OpenNebula.

As a proof of concept for the combination of cloud with cluster network management, we extended OpenNebula with new abstractions and mechanisms that allow to group VMs into virtual clusters and automatically derive a configuration for network isolation from the defined virtual clusters (a configuration file for the *opensm* subnet manager). See Figure 4.1 on page 56 for an overview of the new abstractions and the process that generates the isolation configuration.

The most user-visible extensions are two new abstractions and an interface to control one of them: virtual clusters are groups of VMs that form an organizational unit for allocating network resources, and an abstract topology represents the physical network, using a set of classes that represent elements of a network topology, such as switches, hosts, ports, and links. We extended OpenNebula's XML-RPC interface with operations to create and delete virtual clusters and to associate VMs to them. In addition, we developed a command line tool as an administrative frontend to these operations.

The two new mechanisms operate on the abstract topology and eventually generate a partition configuration, completely automatically without requiring any user interaction. At startup, the OpenNebula core queries all hosts for information about their InfiniBand HCAs. For each HCA, we store the HCA's globally unique identifier (GUID) in the abstract topology inside OpenNebula (in fact, we consider each port of a HCA on its own).

At runtime, our extension periodically generates a configuration file for network isolation. For this purpose, it merges the association of VMs to virtual clusters, the information about which host runs which VM, and the information about the HCAs in each host. The process is simplified because we currently dedicate InfiniBand HCAs to single VMs (because of the restrictions we have described in Section 4.1.2 on page 54). For each virtual cluster, our extension defines an InfiniBand partition that comprises the HCAs of all hosts that run a VM of this virtual cluster. The *opensm* subnet manager is periodically triggered to re-read its configuration file, so that it applies the automatically generated configuration and thereby implements network isolation between the defined virtual clusters. In Section 4.4.1 on page 78, we verify the complete workflow from defining virtual clusters to the automatic generation of partition configuration and check that the actual isolation settings conform with the defined virtual clusters.

In our current prototypic setup, we grant VMs access to an InfiniBand HCA via PCI passthrough. Although OpenNebula does not manage or support device assignment by PCI passthrough by itself, the libvirt [51] toolkit, which OpenNebula uses for starting and controlling VMs based on KVM and qemu, does. We can include configuration statements for libvirt in the definition of a VM in OpenNebula. Such statements are not interpreted by OpenNebula, but just passed on to libvirt unmodified. So, we add libvirt commands that assign the InfiniBand HCA in a host via PCI passthrough to the configuration that defines a VM in OpenNebula, and thereby achieve PCI passthrough access to InfiniBand for automatically deployed VMs. The same way, we instruct libvirt to set CPU affinity for VMs. The configuration allows to set

```
# VM definition for cluster head node
NAME = headnode
# resources
CPU    = 8
VCPU   = 8
MEMORY = 2048
DISK = [ image = "CentOS_5.6", driver = "qcow2" ]

# passed on to libvirt:
RAW = [ TYPE = "kvm", DATA = "
    <devices>
       <hostdev mode='subsystem' type='pci' managed='yes'>
          <source>
            <address bus='0x06' slot='0x00' function='0x0'/>
          </source>
       </hostdev>
    </devices>
    <cputune>
       <vcpupin vcpu='0' cpuset='0'/>
       <vcpupin vcpu='1' cpuset='1'/>
         [...]
    </cputune>
"]
```

Listing 4.1: Exemplary VM definition file for OpenNebula (incomplete).

CPU affinity for each logical CPU of a VM individually, so we can map each logical core to a separate physical core to implement our policy of core dedication. See Listing 4.1 on page 58 for an example VM definition that defines PCI passthrough and CPU affinity.

We prepared a single VM disk image that we use for all virtual cluster nodes. We configured the VM hosts to share OpenNebula's disk image repository via NFS. OpenNebula assigns each VM a fresh copy of the disk image at startup and deletes this copy after the VM has shut down. A VM is instructed via contextualization — that is, the individual parameterization of a VM although it boots from a common disk image — to take the role of a cluster head node, or to become a compute node and thus associate with a given cluster head node (see [47] and Section 3.4.1 on page 49). The VM image contains an *init script* that concludes the boot process by reading the contextualization parameters, and starting the appropriate services. We shall discuss the contextualization mechanism of our prototypic virtual clusters in detail below in Section 4.1.5 on page 58.

4.1.5 Virtual Cluster Environment

In the first three sections on our prototype, we discussed the perspective of the cloud provider, who implements node virtualization, cluster network virtualization, and HPC cloud management to provide users with virtual clusters. In this section, we introduce the environment running inside the virtual cluster, which constitutes the system directly visible to the user. We

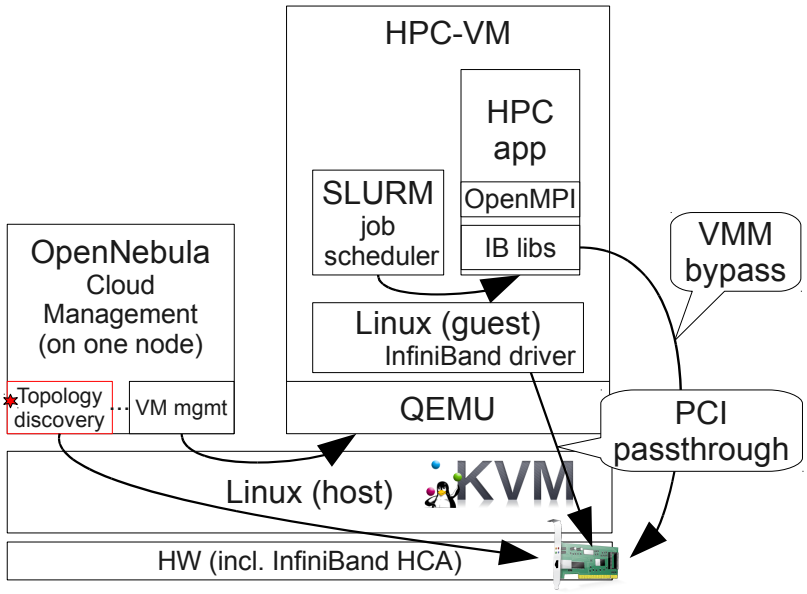

Figure 4.2: Software stack of our prototype HPC cloud, from the host OS up to an HPC application running in a virtual cluster node.

discuss the InfiniBand drivers, libraries, and utilities we employ, the cluster scheduler we used, and how we integrated contextualization to automatically configure a virtual cluster. See Figure 4.2 on page 59 for an overview of the complete software stack on each node.

We installed the OpenFabrics Enterprise Distribution (OFED) version 1.5.3.1 (a bundle of InfiniBand drivers, libraries and tools released by the OpenFabrics Alliance [64]) in the host OSs as well as in the VMs. The OFED bundles more recent driver versions than the vanilla Linux kernel. It further provides all required libraries for InfiniBand access and the MPI implementation OpenMPI, amongst others.

We employed the Simple Linux Utility for Resource Management (SLURM) [98] for cluster batch scheduling. SLURM allocates resources to jobs and manages job queues. It allocates resources in a configurable granularity, reaching from complete nodes (the default) to individual CPU cores or parts of main memory.

OpenNebula supports contextualization to provide VMs with an individual set of parameters, independent from the disk image the VM boots from. For this purpose, it creates an ISO 9660 filesystem image that contains a file with the respective parameters for each VM. We use a single disk image for the virtual cluster's head node and compute nodes and employ the contextualization mechanism to parameterize each node with its role (head or compute node).

The cluster head node mounts an additional persistent data disk image and exports the contained filesystem via NFS (SLURM and MPI jobs require a filesystem that is accessible from all nodes). Further, it starts a DNS server to resolve host names in the virtual cluster. Compute nodes are provided the IP address of the head node as a contextualization parameter. They configure the head node as DNS their server and call a node registration script on the head

node via SSH, providing their host name and IP address as parameters. The node registration script adds the compute node to the head node's configuration — when all compute nodes have booted, the head node has a configuration for SLURM and DNS that comprises all nodes of the virtual cluster, and the virtual cluster is ready.

In our prototype, we installed a similar setup on the host OS of each node, yet with a static configuration in place of contextualization. we use this cluster environment on the hosts to run benchmarks natively and thereby assess baseline performance.

4.2 Node Virtualization

Our prototype comprises components responsible for virtualizing individual cluster nodes, for controlling InfiniBand network access, network isolation and bandwidth usage, for managing the HPC cloud, and for providing an HPC cluster environment ready to submit jobs. So far, we have introduced the components of our prototype. In this section, we evaluate the following issues concerning node virtualization: First, we evaluate how OS background activity (OS noise) is influenced by the features compiled into the Linux kernel and how the different levels of OS noise influence application performance in Section 4.2.1 on page 60. We are especially interested in the configurations that omit features we do not need for HPC workload. Second, we evaluate how time-sharing influences the performance of HPC applications in Section 4.2.2 on page 69. We have examined applications running in VMs and natively on the host OS and observed disproportionate performance degradations from time-sharing, which justify our policy of dedicating physical cores to HPC VMs. Third, we test, whether we can employ Linux scheduling policies to strictly prioritize HPC VMs over VMs with background workload in Section 4.2.3 on page 73. This concept allows background VMs to consume the CPU time left idle by HPC VMs, without disrupting the performance of the HPC applications. That way, we can loosen the strict core dedication policy again, and increase utilization (introducing low-priority background workload instead of consolidating HPC applications).

4.2.1 Minimizing OS Background Activity

Our primary focus in setting up node virtualization is to reduce OS background activity as far as possible, compare Section 3.2.2 on page 26. OS background activity, also referred to as OS noise, influences the performance of HPC applications twofold. First, it reduces the CPU time usable by the application and causes additional cache and TLB pollution. Second, it adds variability to the time an iteration of the application takes on each node, thereby increasing the overall time nodes have to wait for each other (compare Section 2.5 on page 10). To minimize this effect, we disable unnecessary functionality in the host OS by disabling background tasks and by reducing the options compiled into the Linux kernel (or loaded as modules at runtime). In this section, we examine the influence of different compile-time configuration settings of the Linux kernel to determine whether we can achieve a notable reduction of OS noise by employing a custom-configured kernel in the host OS.

We have conducted two series of experiments. First, we used a micro-benchmark for OS noise to examine the influence of individual configuration options, such as disabling swapping support, varying the tick frequency between 100 and 1000 Hz, or enabling dynamic timer ticks. We measured noise in the host OS and determined the best configuration with regard to low OS noise. Second, we ran HPC application macro-benchmarks to assess the influence on actual application performance in a virtualized environment. Based on the results from the first series of experiments, we chose the most promising kernel configurations (there was no single best configuration that generated the least noise in all benchmark runs). We measured application performance with all combinations of these kernel configurations in the host and guest OS.

The remainder of this section is structured as follows: At first, we introduce the benchmarks used. Second, we discuss the results of the synthetic OS noise benchmarks and present the kernel configurations we employed for the subsequent application benchmarks. Third, we present the results of the application benchmarks. Finally, we give a conclusion of our findings.

Benchmarks

We evaluate different kernel configurations in two ways: first, we estimate OS noise, second, we assess the influence on application performance. For this purpose, we employ the Fixed Time Quantum / Fixed Work Quantum (FTQ/FWQ) noise benchmark, and the high-performance linpack (HPL) and the parallel ocean program (POP) macro-benchmarks. We briefly introduce these benchmarks in this section.

We use the Fixed Time Quantum / Fixed Work Quantum (FTQ/FWQ) benchmark from the Advanced Simulation and Computing (ASC) Program's[3] Sequoia Benchmark Codes [10] to analyze OS noise, together with the supplied evaluation script based on the numerical analysis tool GNU Octave. The FWQ benchmark repeatedly measures the execution time of a fixed amount of computation. By comparing all measurements with the minimum measured execution time, one can quantify the system background activity. The minimum execution time is reached, when one iteration runs without interruption by the OS. Other iterations take more time, because they have been disturbed by OS activity. On the contrary, each iteration in the FTQ benchmark runs for a fixed amount of time (the fixed time quantum) and measures how much work can be done in each interval (proposed by Sottile and Minnich in [85]). The FTQ benchmark thereby allows to determine the periodicity of background activities, whereas the FWQ benchmark only permits an overall estimation. Both the FWQ and the FTQ benchmark can run multi-threaded, with each thread pinned to a separate core, to capture the overall behavior of the system.

To assess application performance, we employ two HPC macro-benchmarks, the high-performance linpack benchmark (HPL) [24] and the parallel ocean program (POP) [68]. Both benchmarks use MPI [30] for message-based communication and synchronization between processes. The HPL benchmark solves a system of linear equations using Gaussian elimination. It is used

[3] The *Advanced Simulation and Computing (ASC)* Program of the United States' *National Nuclear Security Administration* utilizes supercomputers for numeric simulations. In the course of the recent acquirement of the supercomputer ASC Sequoia, the ASC has established comprehensive performance criteria and published a set of benchmarks to assess these criteria, the so-called ASC Sequoia Benchmark Codes [9, 10].

to rank supercomputers for the TOP500 supercomputer list [60]. The POP is a fluid-dynamics simulation of ocean currents, which is used for actual research on ocean modeling, not just as a benchmark. The performance characteristics of POP are well studied (see [21, 48]) and POP is often used as an HPC benchmark. Ferreira and associates have found the performance of POP to be very sensitive to OS noise [27], which indeed makes POP a good choice to compare configurations with respect to OS noise.

The HPL benchmark is parameterized by the problem size (the dimension of the randomly generated matrix that defines the set of linear equations to solve) and by the number of processes to employ, amongst others. We fix the problem size to 8000 and vary the number of processes. This setting leads to reasonable execution times with all different numbers of processes we employ. The POP is very flexible and highly configurable, since it is intended for actual scientific simulations. However, there are a few prepared input data sets, which ease the use of POP as a benchmark. We revert to the *X1* benchmark case, which partitions the earth's surface into a grid with a resolution of 1x1 degree, and demands between forty seconds and twenty-four minutes for one run, depending on the number of processes (and CPU cores) POP is distributed to.

Results on OS Noise

In our first series of benchmarks, we have run the FWQ and FTQ noise benchmarks under various kernel configurations in the host OS. We introduce the configurations we tested and the benchmark parameters we employed. We present our measurements, and discuss the configurations we selected for the subsequent application benchmarks.

We used the Linux kernel version 2.6.35 in all our tests. To define the kernel configurations under test, we started with the default configuration provided with the Linux kernel and continuously disabled features. We reduced the timer tick frequency from the default value of 1000 Hz to 100 Hz, switched from continuous timer ticks to dynamic ticks, and disabled swapping. Finally, we disabled all features and drivers we did not directly require. See Table 4.1 on page 62 for an overview of the configurations we tested. This table also introduces the short labels we use to denominate the configurations in the diagrams showing our results.

Label	Description
100 dyn, no swap	defaults, 100 Hz dynamic ticks, swapping support disabled
100 dyn	defaults, 100 Hz, dynamic ticks
100 dyn, min	minimal config, 100 Hz, dynamic ticks
100 static ticks	defaults, 100 Hz ticks
1000 static ticks	defaults, 1000 Hz ticks

Table 4.1: Kernel configurations under test and short labels used in diagrams

We used the FWQ and the FTQ benchmark with two parameter sets that covered a short (about 1 μs) and a long (about 1 ms) interval. For the FWQ benchmark, we specified appropriate values for the number of iterations in each measured work period. For the FTQ benchmark, we specified the desired measurement interval directly. In addition, we varied the number of active

CPUs: In the first run, all 8 CPUs (4 cores with hyper-threading) were active, in the second run, we set all but one CPU offline (via the kernel's *sysfs* interface in */sys/devices/system/cpu/*), so that all OS background activity is concentrated on this single CPU.

We have observed results that differ extremely between the various benchmark runs. There is no single kernel configuration that generates the least OS noise. The variations are clearly visible in Figure 4.3 on page 64, which presents the measured OS noise with short measurement intervals or short work amounts (1 μs), and Figure 4.4 on page 64, which presents the results measured with long intervals and work amounts (1 ms). Tables 4.2 and 4.3 on pages 63 and 63 show the results of the measurements with all 8 active CPUs and with only 1 active CPU in detail. We see that a timer frequency of 1000 Hz causes more OS noise in nearly all of the benchmark scenarios and that a minimal kernel configuration does not result in a minimum of OS background activity, as one might expect.

We created a ranking of configurations to determine the configuration that generates the least OS noise. We derived a relative rating by setting each result in relation to the best achieved result (the least OS noise) in the same benchmark configuration — smaller relative ratings indicate better results. Then, we averaged these relative noise ratings for each kernel configuration to derive a ranking based on OS noise (in addition to an overall ranking, we considered several subsets of the benchmark configurations).

Considering the case of only one active CPU, the minimal kernel configuration creates the least noise in all benchmark configurations. With all eight active CPUs, however, the default configuration without swapping support (100 Hz, dynamic ticks, no swap) generates the least noise over all benchmark runs. Considering the benchmark runs with eight CPUs and a long sample interval alone, the default kernel configuration with 100 Hz tick frequency is slightly better.

configuration	FWQ short	FWQ long	FTQ 1μs	FTQ 1ms
1000 Hz, static ticks	28.9%	18.08%	12.6%	3.36%
100 Hz, static ticks	12.0%	3.39%	11.6%	0.77%
100 Hz, dynamic ticks	11.9%	0.70%	10.3%	0.17%
100 Hz dynamic ticks, no swap	14.2%	0.22%	12.5%	0.18%
100 Hz dynamic ticks, minimal	13.1%	0.43%	14.2%	0.32%

Table 4.2: Detailed Results of OS noise measurements with all 8 CPUs active.

configuration	FWQ short	FWQ long	FTQ 1μs	FTQ 1ms
1000 Hz, static ticks	10.5%	0.20%	11.4%	0.18%
100 Hz, static ticks	11.8%	0.11%	11.1%	0.09%
100 Hz, dynamic ticks	12.2%	0.11%	10.8%	0.09%
100 Hz dynamic ticks, no swap	10.2%	0.11%	9.3%	0.10%
100 Hz dynamic ticks, minimal	7.2%	0.10%	10.8%	0.09%

Table 4.3: Detailed Results of OS noise measurements with only 1 CPU active.

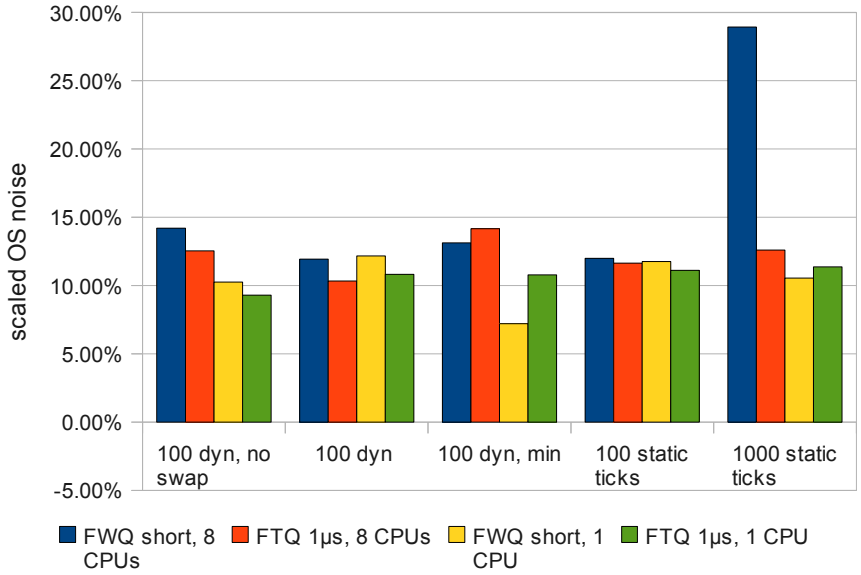

Figure 4.3: Measured OS noise with short work amounts and sample intervals (about 1 μs).

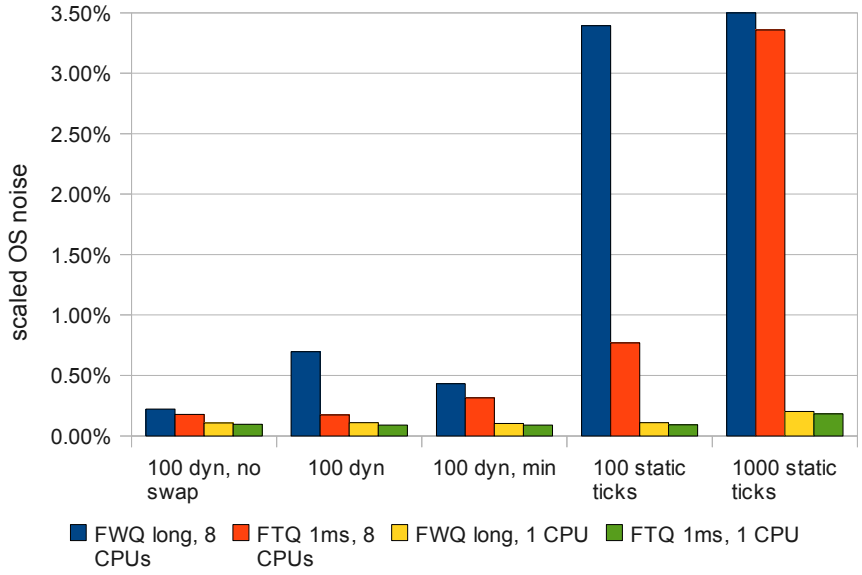

Figure 4.4: Measured OS noise with long work amounts and sample intervals (about 1 ms). The result of FWQ in the configuration with 1000 Hz timer frequency (28.9%) is clipped.

Since the measurements with the synthetic noise benchmarks FWQ and FTQ did not indicate a single best configuration with regard to OS noise, we chose the three kernel configurations that each generated the least OS noise in a subset of the benchmark settings. We shall continue the evaluation with application benchmarks using (1) the default configuration with a 100 Hz timer frequency and dynamic ticks, (2) the same configuration without support for swapping, and (3) the minimal configuration with 100 Hz timer frequency and dynamic ticks.

Results on Application Performance

In our first series of experiments, we have run a synthetic OS noise benchmark on the host OS and varied the configuration of the Linux kernel. There is no single configuration that generated the least OS noise in all benchmark runs, but we identified three candidate configurations. In this section, we continue the evaluation of OS noise with application benchmarks we run natively and in VMs with all combinations of the candidate kernel configurations in the guest and host OS. First, we discuss the scalability of the benchmark applications on our test cluster hardware (see Section 4.2.1 on page 61 for an introduction to the benchmarks) and derive corner stone benchmark configurations (the number of processes and their distribution to nodes) for use in the subsequent evaluation. Second, we present the results of the actual benchmark runs. Throughout these experiments, we disabled the Intel Turbo Boost feature, which dynamically scales up the CPU frequency depending on the load of individual cores. That way, we avoid a negative influence on scalability caused by the reduction of the CPU frequency as we utilize more cores.

We intended to run each benchmark ten times using each possible benchmark configuration, under each combination of host and guest OS kernel. However, a single iteration through all benchmark configurations, from one process on a single node up to 32 processes distributed over four nodes, took an impractical amount of time. So, we decided to focus on a few benchmark configurations we selected by considering the scalability of the benchmark applications — it does not make sense to benchmark inefficient configurations without practical relevance. In Figure 4.5 on page 66 we depict the scalability of the high-performance linpack (HPL) benchmark and the parallel ocean program (POP). We present the speedup from employing more processes per node (up to one process per hardware thread) and from employing more nodes, relative to the case of a single process on a single node.

We clearly see that hyper-threading — that is, running more than four processes on each node — only benefits performance when we employ it uniformly on all cores by running eight processes per node. Otherwise, with five to seven processes per node, performance decreases.

With up to four processes per node, each added process opens up the compute resources of a new physical core. With five to seven processes per node, some of the processes contend for the execution units of a physical CPU core, because they run on hardware threads of the same core. Therefore, the processes run at different speed and some have to wait in each iteration. With eight processes per node, however, each available hardware thread is utilized and the processes run at nearly uniform speed. This effect is much more pronounced with the POP benchmark.

We can also deduce from Figure 4.5 on page 66 that HPL is memory-bound on our test cluster, because the performance gain levels out beyond three processes per node — all four cores

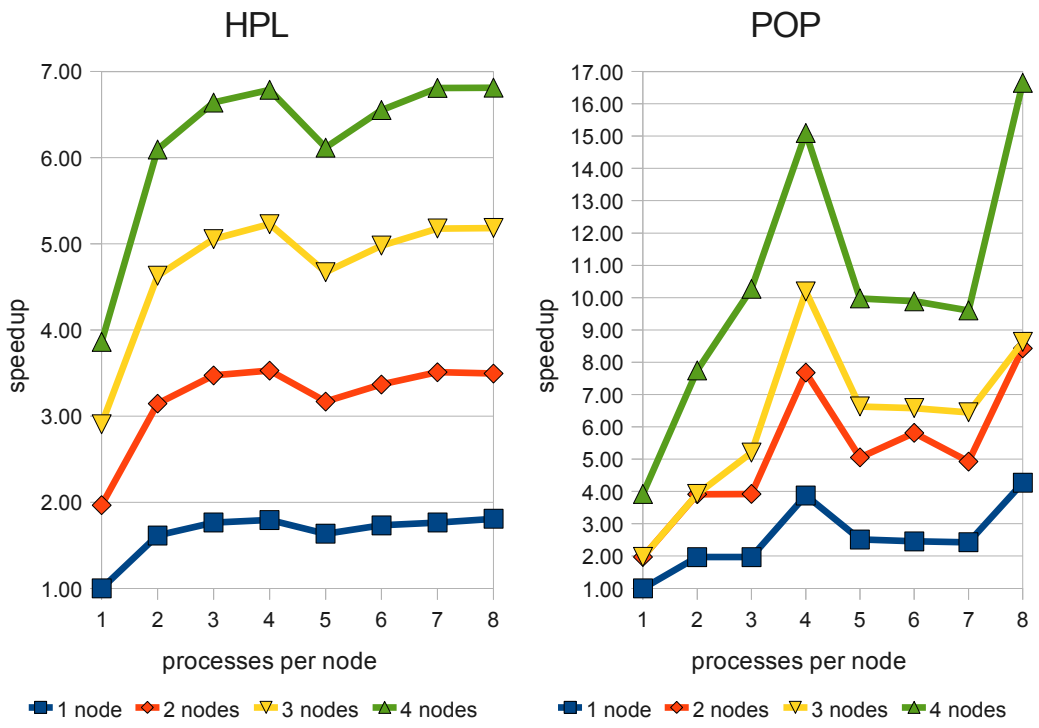

Figure 4.5: Scalability of the HPL and POP benchmarks on our test cluster. Hyper-threading is effective with five to eight processes per node.

are in a single chip and share L3 cache and memory controller. POP however scales well with additional cores per node — on a single node, we observed almost linear speedup with four cores (3.9). POP is constrained by its problem decomposition however, which causes unexpected results when the number of processes is not a power of two. For example, three processes on a single node achieve no more performance than two processes , whereas the step from two to three processes per node causes a performance gain when three or four nodes are involved.

Based on these observations, we chose a subset of all possible benchmark configurations for the comparison of the candidate kernel configurations. We run the benchmarks with one and with four nodes to capture the minimum and the maximum virtual cluster configuration possible in our test environment. We varied the number of processes per node between one, two, four, and eight, to capture the observed performance corner stones (one, four, and eight processes) and additional information about scalability (two processes). We eliminated five to seven processes, because they pose a performance loss compared to using four processes.

We aggregated the individual results to determine, which combination of host/guest OS kernels provides the best application performance. We determined the best result in each benchmark combination, achieved in any combination of host/guest OS kernels (including the native benchmark runs in the host OS). Then, we calculated the relative performance penalty of each

host/guest OS kernel combination (caused by virtualization overhead and different kernel configurations), separate for each benchmark configuration and averaged over all of them.

We present the relative performance penalty, averaged over all benchmarks configurations in Table 4.4 on page 67 for the HPL benchmark, and in Table 4.5 on page 67 for the POP benchmark. We show a performance penalty for each of the kernel configurations, because we chose the minimum over all configurations as the reference performance level and none of the kernel configurations accounted for all best results.

		Guest		
Host	Native	100 Hz, dynamic ticks	100 Hz, dynamic ticks, no swap	100 Hz, dynamic ticks, minimal
100 Hz, dynamic ticks	1.5%	4.3%	3.9%	3.7%
100 Hz, dynamic ticks, no swap	0.1%	3.9%	4.1%	3.8%
100 Hz, dynamic ticks, minimal	0.2%	9.3%	9.4%	9.5%

Table 4.4: Relative performance penalty of different configurations of host and guest OS kernel for the HPL benchmark. Reference performance values are the best observed results in each benchmark configuration.

		Guest		
Host	Native	100 Hz, dynamic ticks	100 Hz, dynamic ticks, no swap	100 Hz, dynamic ticks, minimal
100 Hz, dynamic ticks	0.1%	2.0%	2.0%	1.8%
100 Hz, dynamic ticks, no swap	0.04%	1.9%	1.8%	2.3%
100 Hz, dynamic ticks, minimal	0.2%	20.2%	21.3%	21.1%

Table 4.5: Relative performance penalty of different configurations of host and guest OS kernel for the POP benchmark. Reference performance values are the best observed results in each benchmark configuration.

Obviously, the minimal kernel configuration is a bad choice for the host OS kernel (we have cross-checked that we did not disable any features required for virtualization by accident). Although both POP and HPL perform well running natively in a minimal kernel, this kernel causes significant overhead with virtualization (from more than a factor of two for the HPL benchmark to an order of magnitude for POP).

Considering the native case alone, the configuration with disabled swapping support (defaults with 100 Hz timer frequency, dynamic ticks, and disabled swapping support) performs best.

With virtualization on top however, the results between the configurations with and without swapping support (otherwise the same) do not show differences to a statistically significant degree. Surprisingly, the configuration of the guest OS kernel seems to be irrelevant (again, no statistically significant differences).

	processess per node			
nodes	1	2	4	8
1	1.2%	1.4%	1.7%	1.3%
4	0.8%	2.0%	2.2%	3.4%

Table 4.6: Virtualization overhead of POP with the kernel configuration without swapping.

	processess per node			
nodes	1	2	4	8
1	2.9%	3.5%	1.3%	1.5%
4	4.01%	4.9%	3.9%	10.2%

Table 4.7: Virtualization overhead of HPL with the kernel configuration without swapping.

Independent from the kernel configurations (leaving out the minimal configuration), we have observed acceptable virtualization overheads. As an example, we show the virtualization overhead of the configuration with disabled swapping support for POP in Table 4.6 on page 68, and for the HPL benchmark in Table 4.7 on page 68 — the results for the other combinations, besides those with a minimally configured host OS kernel, are in the same range. We observed surprisingly low overheads for the POP: Virtualization overhead for the POP is below 2 % in most cases, and below 3.5 % in the most distributed configuration. The HPL benchmark suffers much greater overhead from virtualization: We observed a virtualization overhead for the HPL benchmark of up to 3.5 % on a single node, up to 5 % with one to four processes each on four nodes, and 10 % in the most distributed case of four nodes with eight processes each.

We attribute the difference in virtualization overhead between POP and HPL benchmark to their different characteristics, which we discussed in this Section. One explanation is that the POP is very CPU-intensive and performs only few memory access operations (in relation to HPL), whereas the HPL benchmark is very memory-intensive (we have observed that two processes nearly saturate the memory bus in our test hardware). Executing instructions on the CPU in user level, without memory accesses, has no virtualization overhead. In contrast, address translation suffers the overhead of nested paging with hardware virtualization (as used by KVM), compare [1]. Being much more memory-intensive, HPL causes much more TLB misses with virtualization overhead.

We cannot yet assess the influence of InfiniBand virtualization overhead on the two benchmark applications. Performance studies that attest POP a fine-grained communication pattern [21, 48], and our own observations of very infrequent communication in the HPL benchmark (derived with an MPI profiling tool) suggest, however, that POP is much more prone to virtualization overhead on communication operations.

Summary

We have measured OS noise in various kernel configurations and assessed the influence on application performance in different combinations of host and guest OS kernels. We employed the synthetic OS noise benchmarks FWQ and FTQ from the ASC Program's Sequoia Benchmark Codes [10] and the macro-benchmarks HPL and POP.

The FWQ and FTQ benchmarks indicated the default kernel configuration with 100 Hz timer frequency and dynamic ticks, the same configuration with disabled swapping support, and a minimal kernel configuration as candidates for low OS noise — no single configuration provided the lowest noise levels in all benchmark configurations. They do all miss the specifications of the ASC Program and the request put up in [89], however. The ASC Sequoia Request for Proposals defines a *diminutive noise environment* based on the results of the *Fixed Work Quantum* benchmark: a runtime environment is called a *diminutive noise environment,* if the OS noise measured by the FWQ benchmark has a scaled mean of $< 10^{-6}$, amongst further constraints on the statistics of the noise (compare section 3.2.4 in Attachment 2, Draft Statement of Work, in [9]). Tsafrir and associates deduced in [89], that OS noise should be in the order of 10^{-5} for clusters with thousands of nodes. Yet, our results show scaled noise means in the range of 10^{-4} to 10^{-2}.

We tested the candidate kernel configurations as host and guest OS kernel in all combinations. We can clearly eliminate the minimal configuration as a feasible host OS kernel, because it has significantly greater overhead than the alternatives. The other two configurations (default, 100 Hz timer frequency, dynamic ticks, with/without swapping support) did not show a statistically significant difference when used as host OS. The configuration without swapping support is slightly advantageous in the native case, however. Surprisingly, we did not observe significant performance differences when varying the guest OS kernel — the host OS kernel proved to be the more relevant influence. Virtualization overhead, as observed in our experiments, is acceptable for both benchmark applications.

Combining the results of both series of experiments, we conclude that there are significant influences on application performance besides OS noise alone. A completely minimalistic kernel configuration is not a good choice for the host OS and provides significantly worse performance than the other tested configurations, although it generated low OS noise levels, too. The default kernel configuration provides better performance with virtualization, and disabling swapping support potentially brings a slight advantage.

4.2.2 Influence of Time-Sharing

Our approach sets two basic CPU allocation policies: we dedicate physical cores to logical cores for HPC VMs, instead of time-sharing physical CPU cores between logical cores for HPC workload, and we employ non-HPC VMs to absorb CPU time left idle by HPC jobs and prioritize them strictly lower than HPC VMs. We shall evaluate Linux scheduling policies for prioritizing VMs in the following Section 4.2.3 on page 73. In this section, we will justify the policy of dedicating cores, by showing that the performance of HPC applications suffers

disproportionately from timesharing, especially when fine-grained communication and synchronization between several processes is involved. We introduce the experimental setup and the benchmark applications we use, describe the communication scenarios we compare, and present the results of our measurements. Finally, we contrast lost performance with resources saved by timesharing (CPU cores). As introduced in the previous section, we disabled the Intel Turbo Boost feature of the CPUs to avoid the negative influence on scalability caused by the reduced CPU frequency when more cores are utilized.

We run the experiments on timesharing on our HPC cloud prototype, which we introduced above in Section 4.1 on page 53. We disable the Hyper-Threading[4] feature of the CPUs — that is, symmetric multi-threading — to avoid the influence of resource contention between hardware threads. We start two VMs per host, each with 4 logical CPU cores, and configure CPU affinity, such that each logical core of a VM shares a physical core with a logical core of another VM. We use four VMs for measurements (one per host), and the other four VMs to generate background workload that shares the CPU.

We employ the HPC macro-benchmarks high-performance linpack (HPL) and parallel ocean program (POP), which we have introduced already in Section 4.2.1 on page 61. Both benchmark programs resemble the bulk-synchronous single-program multiple-data (SPMD) application model, which we have introduced in Section 3.1 on page 17. Figure 2.1 on page 11 depicts the two phases a bulk-synchronous SPMD application cycles through: local computation alternates with synchronization and communication between all nodes.

We compare four scenarios that differ in the number of processes and their distribution to nodes, and, consequently, in the type of communication involved. We run each benchmark as (1) a single process (no communication at all), (2) four processes on the same node (node-local communication), (3) four processes distributed to four nodes (remote communication), and (4) sixteen processes equally distributed over four nodes (local and remote communication). In each scenario, we compare performance between idle and fully loaded background VMs, resembling dedicated cores and timesharing. To generate the background load, we repeatedly run the HPL benchmark in each background VM (automated by a shell script, with node-local communication only), which completely utilizes the logical cores.

scenario	slowdown by factor	
	HPL	POP
(1) 1 process on 1 node	3.0	2.2
(2) 4 processes on 1 node	2.0	2.9
(3) 4 processes on 4 nodes	3.3	2.7
(4) 16 processes on 4 nodes	2.9	11.5

Table 4.8: Slowdown of HPL and POP benchmarks in a virtual cluster caused by timesharing the CPU with a fully loaded VM.

[4] Hyper-Threading is the denomination of the symmetric multi-threading (SMT) capability of Intel CPUs, which allows two threads of execution, each with a complete register set on its own, to share the functional units of a CPU core [55].

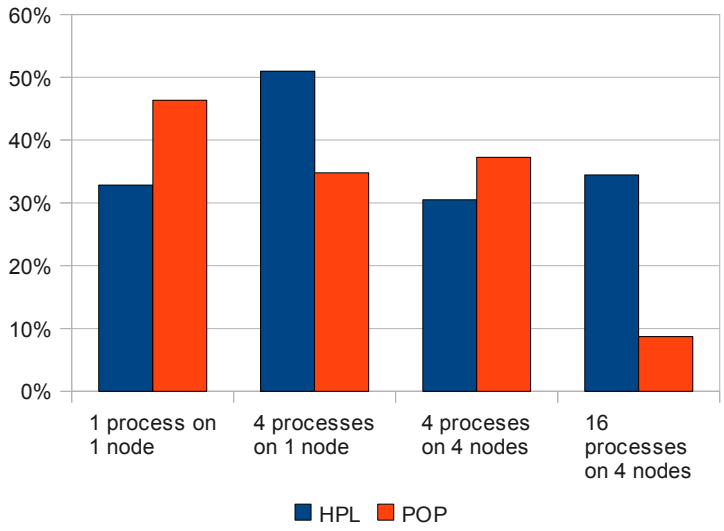

Figure 4.6: Remaining performance of the HPL and POP benchmarks with timesharing in various process distribution and communication scenarios.

Clearly, one expects a reduction of performance when the VM running an HPC benchmark receives only half the CPU time. In addition to assigning less CPU time, timesharing causes more context switches and thus cache misses, which all add overhead and further reduce performance. We present our results in Table 4.8 on page 70, which lists the factors of slowdown by timesharing, and Figure 4.6 on page 71, which indicates the remaining performance with timesharing relative to the case without timesharing, as an alternative display. In most instances, we have observed a much larger slowdown than only by a factor of two (by cutting assigned CPU time in half), especially when remote communication between nodes is involved.

As the HPL and POP benchmarks behave differently, with HPL being more memory-mound than POP, and POP employing more fine-grained communication than HPL (as we already observed in Section 4.2.1 on page 65), they are consequently affected differently by timesharing. So, we discuss our observations for each benchmark separately, at first. Then, we draw a conclusion over both cases.

The HPL benchmark suffers a slowdown of around a factor of three in most scenarios. This slowdown appears already with only one process and one CPU core involved, as in scenario (1), and does not increase significantly with the introduction of remote communication, as in scenarios (3) and (4). We deduce, that the influence of timesharing on the HPL benchmark is mainly caused by local contention for memory bandwidth, whereas the course-grained communication pattern does not suffer much from jitter introduced by timesharing. The slowdown with four processes on a local node, see scenario (2), agrees with this observation. Performance with four processes per node is already limited by memory bandwidth without timesharing, not by execution speed. So, reducing CPU time to one half does not introduce additional adverse cache effects and just cuts performance to one half.

In contrast, performance of the POP benchmark scales with the number of cores, but is heavily influenced by its fine-grained communication. As a result, the slowdown of a single process caused by timesharing, scenario (1), is not much greater than the reciprocal of the reduction of CPU time. However, when the POP benchmark is distributed over several processes that communicate and synchronize with each other, timesharing increases jitter on the execution time of each process and thereby increases the overall time processes have to wait for each other. See Section 3.2.2 on page 26 for a description of this effect. As a result, performance is reduced by more than a factor of ten in the most distributed case we tested — that is, scenario (4) with sixteen processes that communicate locally and remotely.

For the purpose of cross-checking our results from the virtualized environment, we repeated our measurements with the benchmarks running natively as processes on the host OS (reduced to the extreme cases scenario (1) and (4) in the POP benchmark). We employed CPU affinity (supported by OpenMPI) to cause each CPU core to be shared between two benchmark processes (one of them as background workload), as with VMs before. In this setting, we observed slowdowns very similar to the virtualized case, see Table 4.9 on page 72. Thus, we can exclude that the observed behavior is caused by virtualization only.

scenario	slowdown by factor	
	HPL	POP
(1) 1 process on 1 node	3.9	2.2
(2) 4 processes on 1 node	2.0	-
(3) 4 processes on 4 nodes	4.2	-
(4) 16 processes on 4 nodes	2.9	10.1

Table 4.9: Slowdown of HPL and POP benchmarks in the native case, caused by timesharing the CPU with another benchmark process on a physical node.

In summary, timesharing reduces the performance of the HPC benchmark applications we examined disproportionately. We added a second HPC workload that shared CPU time in a 1:1 relation, but in most cases observed a reduction of performance to about 30% (remarkably below 50%). Timesharing is intended to better utilize compute resources by putting otherwise unavoidable idle time to use, but with HPC applications as we experienced, it does not pay off: We employ half the compute resources per application (50% CPU time) but achieve only one third of the performance. To process a given amount of work, we effectively require more compute resources with timesharing, because we achieve less work per CPU time.

This result supports our decision to dedicate physical cores to VMs for HPC workload, because the alternative, timesharing a physical CPU between several VMs, reduces performance in a disproportionate way. In the following section, we examine a scenario that looks similar to timesharing as analyzed in this section, but replaces the symmetric setting of timesharing between several equal HPC VMs with an asymmetric relation of a prioritized HPC VM and a subordinate background VM that may only consume CPU time explicitly left idle.

4.2.3 Linux Realtime and Idle Scheduling Policies

In the previous Section 4.2.2 on page 69, we have examined the influence of timesharing on equally prioritized parallel applications. We have seen that sharing a physical core between parallel applications reduces performance considerably more than expected. Thus, we avoid timesharing, but try to increase utilization by running a low-priority background VM on each physical core, in addition to a high-priority VM for HPC workload, see Section 3.2.2 on page 26. For this purpose, we have to take care that a non-HPC VM only receives CPU time left idle by HPC VMs. As introduced in the previous section, we disabled the Intel Turbo Boost feature of the CPUs to avoid the negative influence on scalability caused by the reduced CPU frequency when more cores are utilized.

On each node of the physical cluster that forms our HPC cloud, we employ Linux and KVM for virtualization. With this virtualization stack, a VM is scheduled like a regular process in the host Linux (see Section 3.2.1 on page 21). Regular dynamic priorities with *nice* levels do not suffice, because they cannot strictly prioritize one process above another (compare [86]). Instead, we intended to employ static (soft) realtime priorities[5] and assign HPC VMs a higher priority than non-HPC VMs (with cloud management tasks prioritized even above). However, we have found this approach to be infeasible with PCI passthrough access to the InfiniBand HCA, because the drivers in the VM experienced timeouts when accessing the HCA, causing HPC applications to fail during initial connection setup. In addition, measurements of OS noise (as we use also in Section 4.2.1 on page 60) have shown increased OS background activity when the VM or the noise benchmark process in the host OS was assigned a static realtime priority.

In place of static realtime priorities, we revert to assigning a non-HPC VM the *SCHED_IDLE* scheduling policy, causing it to be scheduled only when no other tasks are ready and, thereby, being prioritized below the HPC VMs. In the remainder of this section, we examine whether we can practically assign a VM the idle scheduling policy and whether assigning this policy really prevents non-HPC VMs from disturbing HPC VMs. For this purpose, we repeated a subset of the measurements from the previous Section 4.2.2 on page 69, this time however with the SCHED_IDLE policy assigned to the background VM. Again, we generate the background load by repeatedly running the HPL benchmark in each background VM and disable hyper-threading to avoid resource contention inside a physical CPU core. HPL is very memory-intensive and thereby constitutes a very resource demanding background workload.

We present the results of our measurements in Table 4.10 on page 74, which show the factors of slowdown by a running background VM, and Figure 4.7 on page 74, which indicates the relative remaining performance.

The POP is influenced only minimally in all scenarios. The loss of performance is less than 1.5% in the most distributed case we tested (16 processes on 4 nodes), and significantly less than 1% in all other cases. This overhead is acceptable and scheduling of HPC and background VMs works as expected.

[5] The system call *sched_setscheduler* defines the scheduling policy used for a process. *SCHED_FIFO* and *SCHED_RR* (round-robin) are the two supported (soft) realtime policies. See also [57] and the *manual page* of the *sched_setscheduler* system call.

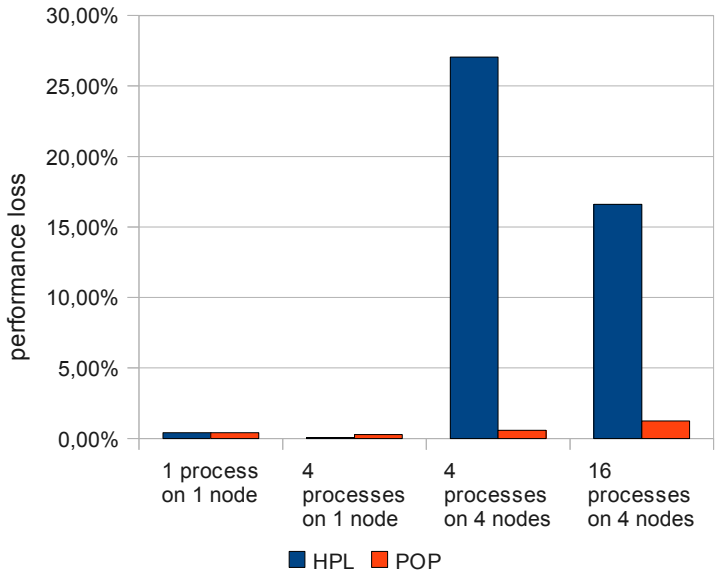

Figure 4.7: Performance loss of the HPL and POP benchmarks caused by low-priority background VMs with a memory-intensive workload.

scenario	slowdown by factor	
	HPL	POP
(1) 1 process on 1 node	1.004	1.004
(2) 4 processes on 1 node	1.001	1.003
(3) 4 processes on 4 nodes	1.371	1.006
(4) 16 processes on 4 nodes	1.199	1.012

Table 4.10: Slowdown of HPL and POP benchmarks in a virtual cluster caused by a low-priority background VM.

The HPL benchmark however, suffers from contention for memory in scenario (3). There, the high-priority VM leaves three logical cores idle, so the background VM occupies the respective physical cores. As a result, all four cores run instances of the HPL benchmark and contend for the memory bus.

The performance loss of the HPL benchmark in scenario (4) has a slightly different reason. In this scenario, the HPL benchmark uses all four logical cores of the benchmark VM and therefore blocks out the background VM most of the time. So, one might expect no performance loss at all. There are short periods however, during which the HPL benchmark leaves the CPU idle: Earlier profiling runs have shown that the communication pattern of the HPL benchmark causes OpenMPI to use blocking send and receive operations. In these periods, the background VM and thus the background HPL benchmark inside is allowed to run, and consequently trashes the cache. When the observed high-priority HPL benchmark resumes, this effect causes compulsory cache misses that do not occur with the HPL benchmark using the

CPU alone.

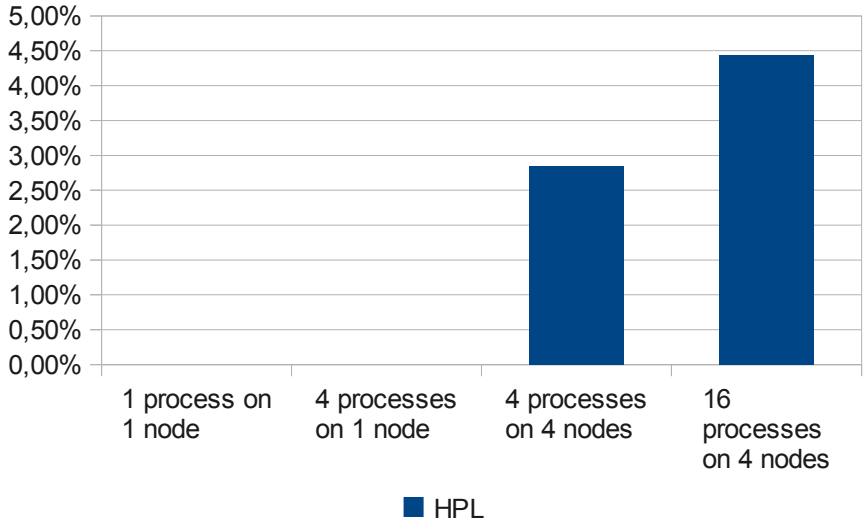

Figure 4.8: Performance loss of the HPL benchmark caused by low-priority background VMs with a CPU-intensive workload.

We have cross-checked the behavior of the HPL benchmark with a CPU-intensive background workload that performs hardly any memory accesses (the FWQ OS noise benchmark). See Figure 4.8 on page 75 for the results. Running the benchmark on a single node, we could not measure a difference in performance. Running the benchmark distributed over four nodes, thus with communication and the aforementioned short interruption periods involved, we observe a reduction in performance of up to 4.5%. These results are remarkably different from the measurements with a memory-intensive background workload and confirm our observations regarding contention for memory bandwidth and cache effects due to short interruptions.

In conclusion, we achieve strict prioritizing of HPC VMs over background VMs regarding CPU time by assigning background VMs the SCHED_IDLE policy. CPU-bound workloads, such as the POP benchmark, experience only small reductions of performance when the background workload is active. Contention for other resources than CPU time, which are beyond control of host OS scheduling, can reduce performance for memory-bound workloads however.

4.3 Network Virtualization

In this section, we evaluate a basic building block of our HPC cloud architecture. To guarantee network isolation, the HPC cloud provider must keep the configuration of the physical network under exclusive control. For this purpose, the InfiniBand management interfaces must be protected against unauthorized modifications. We evaluate the practical feasibility of a key-based protection scheme in Section 4.3.1 on page 76.

4.3.1 Protection of Management Interfaces and Isolation

One of our primary design goals for network virtualization is to provide a user with a virtual view of his share of the InfiniBand network, which he may configure just like a physical network, as we described in Section 3.3.1 on page 31. At the same time, we have to prevent a user from tampering with the configuration of the actual physical InfiniBand network of the HPC cloud, so that he cannot compromise isolation or modify bandwidth allocations (compare Section 3.3.3 on page 38). So, on the one hand, we have to lock out a user from access to the management interface of the physical network. Yet on the other hand, we virtualize the InfiniBand management interface and thereby allow a user to configure his share of the network using this interface.

In this section, we briefly recapitulate the two mechanisms we employ to protect the management interface of the physical InfiniBand network and thereby isolation, and evaluate the practical applicability of one of them. We show that the tool we employ for the setup of InfiniBand network isolation and QoS, the *opensm* subnet manager, fails to implement protection. We describe the necessary corrections we applied to make protection work correctly. Finally, we discuss the protection level that our modifications accomplish and describe one further required improvement. In addition to conceptual consideration, we based this evaluation on experiments with actual hardware.

InfiniBand network isolation is configured by partitioning the network into groups of hosts (partitions) and by restricting nodes to communicate only within these groups, see Section 3.3.2 on page 35. The assignment of nodes to partitions as well as the partition enforcement mechanism, which filters packets while they pass a switch, are both configured using the InfiniBand subnet management protocol. Thus, protecting isolation against illegitimate modifications means protecting the configuration of the physical InfiniBand network against modification. For this purpose, we employ two mechanisms: We intercept management datagrams sent by a VM, and we enable the management protection mechanism defined by InfiniBand. In this section, we verify by experimentation that the InfiniBand management interfaces, and thus isolation, can be effectively protected against modifications using protection keys.

As we introduced in Section 3.3.3 on page 38, the protection key mechanism, once enabled, causes a node (actually, the subnet management agent of the node) to accept only those management and configuration requests that state the correct key. Thus, only the subnet managers that know this key (primarily the one that assigned it) can alter the network configuration after protection has been enabled. The SR-IOV compatibility issues on our test hardware kept us from experiments with the exchange of management datagrams between host and VM (and thus, possibly, interception), employing SR-IOV. Since we cannot alternatively publish a code review prior to the official release of the SR-IOV drivers, we have to leave this topic to future work and focus on protection keys here.

In our architecture of an HPC cloud, the cloud provider employs a subnet manager to configure isolation and bandwidth shares. To prevent users that accidentally (or forcefully) gained complete access to a HCA, and thus can send management packets into the physical network, from modifying this configuration (e.g., to violate isolation), the cloud provider enables the protec-

tion key mechanism to lock out any other subnet manager besides his own (compare Section 3.3.3 on page 37 and Section 3.3.4 on page 44).

We run two subnet managers on different nodes at the same time and observe, which of them successfully influences the subnet configuration. One resembles the cloud provider who tries to protect the configuration, the other one depicts a malicious user that tries to circumvent isolation. In addition, we employed

- the *ibstat* utility, to determine node addresses and the currently registered subnet manager on every node;
- the *ibnetdiscover* utility to query the network topology, a read-only access to the InfiniBand management interface;
- and the *smpquery* utility to retrieve all settings related to subnet management from a node.

All these utilities are contained in the package *infiniband-diags* of the OpenFabrics Enterprise Distribution (OFED).

Without subnet management protection, we can start two subnet managers in succession and cause the second one to simply overwrite the settings of the first (calling *opensm -d 0 -r*). The second subnet manager assigns fresh node addresses (LIDs), registers itself as the current subnet manager with every node (verified with the *ibstat* utility) and can freely reconfigure isolation (including to disable it completely). Further, we can employ the *ibnetdiscover* utility to determine the topology of the complete physical subnet and use the *smpquery* utility to query all settings.

When we try to enable protection with a regular *opensm* subnet manager, by configuring it to use use a protection key, the actual level of protection does not change. Another subnet manager can still change settings freely, we can query or modify isolation settings, and we can even readout the ineffective protection key or the isolation settings (partition keys) from any node using the *smpquery* utility. This lack of protection is a result of the failure of current versions[6] of *opensm* to actually enable the protection mechanism in a node (by setting the protection flags after storing the protection key). In addition, *opensm* currently supports only one protection key for all nodes in the network, which compromises the complete network, when the protection key has been revealed on a single node[7].

We modified *opensm* to set the protection flags in each node: We changed the data structure used to construct management datagrams, so that it includes the subnet management protection flags; and we extended the initial configuration of each node (assigning an address, registering the subnet manager, etc.) to also enable the protection flags in this step. As a result, we achieve

[6] We based our experiments on opensm version 3.3.9 as included in the OFED version 1.5.3.1. However, a cross-check with the current development tree (as of 27 May 2011, retrieved from `git://git.openfabrics.org/~alexnetes/opensm.git`) showed the same problem with regard to protection.

[7] Using the same protection key on all nodes in the network enables the following attack: If a malicious user gains complete access to an InfiniBand HCA by breaking out of his VM, he may be able to extract the protection key from this HCA (it is possible with ConnectX-2 HCAs). The single extracted protection key would then allow to modify the configuration of all other nodes. To circumvent this attack, we assign each node a different protection key.

the protection of the management interface of the physical InfiniBand network as intended. As long as the protection key is kept secret and no one guesses it or tries all possible keys (with a small possibility of success because of 64 bits key length, compare Section 3.3.3 on page 38), we can be sure that

- no other subnet manager, besides the one employed by the cloud provider, can modify the configuration of the physical network, including the configured network isolation;
- the topology of the (whole) physical network cannot be determined (e.g., using the utility *ibnetdiscover*), because nodes do not answer unauthorized discovery requests;
- protection keys and isolation settings (partition keys) cannot be readout, because the get requests will be discarded without the correct key.

We verified these three claims by experimentation in our test cluster with VMs that have complete access to the InfiniBand HCA via PCI passthrough (our prototype setup, see Section 4.1.2 on page 54). In summary, a user that breaks out off his VM and gains complete control over the InfiniBand HCA cannot use this gain to tamper with the physical InfiniBand network. Of course, these restrictions do not apply to a user who legitimately utilizes the virtualized management interface to configure his share of the network, as we describe in Section 3.3.1 on page 31.

One limitation of *opensm* still remains: It can currently assign only one protection key to all nodes in the network. In a second modification attempt, we changed the assignment of protection keys to generate a unique key for each node. However, this attempt is still early work and currently inoperative because some data structures are used inconsistently in *opensm* and we thus cannot retrieve the assigned protection key for future accesses to a node. Nonetheless, we achieved that protection keys are employed effectively at all.

4.4 HPC Cloud Management

So far, we have evaluated policies we have chosen and mechanisms we employ in the areas of node and network virtualization. At last, we examine HPC cloud management in this section. First, we verify the automatic generation of an InfiniBand network isolation configuration for virtual clusters by following the complete workflow from defining virtual clusters to checking that isolation has been established as desired. Second, we examine the scalability of OpenNebula and discuss two potential bottlenecks.

4.4.1 Automatic Configuration of Network Isolation

In our HPC cloud architecture, cloud management is in charge of combining and orchestrating node and network virtualization. In the previous sections of our evaluation, we have examined VMs with access to InfiniBand HCAs on the one hand, and isolation in an InfiniBand network on the other hand — however, each aspect separately. In this section, we examine how they work together under the control of HPC cloud management: We follow the workflow of our prototypic extension to OpenNebula that automatically configures network isolation for virtual

```
# define a virtual cluster with 2 nodes
$ onevcluster create vc1
$ onevcluster addheadnode     vc1 255
$ onevcluster addcomputenode vc1 256

# define a second virtual cluster with 1 node
$ onevcluster create vc2
$ onevcluster addheadnode vc2 257

# query virtual clusters (NODEC = node count)
$ onevcluster list
   ID      NAME STAT NODEC HEADN
    1       vc1    1     2   255
    2       vc2    1     1   257

# query running VMs
$ onevm list
   ID      USER      NAME VCLUSTER STAT CPU      MEM   [...]
  255     admin  headnode       vc1 runn   0       0K
  256     admin  compnde1       vc1 runn   0       0K
  257     admin  headnde2       vc2 runn   0       0K
```

Listing 4.2: Configuring virtual clusters in OpenNebula.

clusters, as we introduced in Section 4.1.4 on page 56. First, we describe the experimental setup of this test and the involved real and virtual nodes and their roles. Second, we present how we actually tested the automatically configured isolation by checking isolation between two virtual clusters.

As stated before, SR-IOV is currently not working in our prototype and we have to dedicate complete InfiniBand HCAs to VMs. We thereby loose InfiniBand access in the respective host OS. Since we need InfiniBand access in the host OS running the OpenNebula core to configure network isolation, we could not use this host for VMs in this test. As a result, we used a virtual cluster consisting of two nodes, and another virtual cluster consisting of only one node.

We associated VMs to virtual clusters using our newly developed command line tool, which utilizes the new methods for virtual cluster management in OpenNebula's XML-RPC interface, as we introduced in Section 4.1.4 on page 56. In Listing 4.2 on page 79 we present the issued commands and the resulting virtual cluster configuration in detail.

Our OpenNebula extension had queried all HCAs in the hosts for their unique ids while the hosts still had access to the InfiniBand HCAs before VMs had been started. At runtime, it periodically generated a partition configuration from the queried unique ids and from the virtual clusters defined by us. We added a reference to the generated partition configuration to the main configuration file of the *opensm* subnet manager, to make *opensm* incorporate the partitions into the generation of the physical InfiniBand network. See Listing 4.3 on page 80 for an example of a generated partition configuration,

We checked the successful setup of isolation with a modified version of an InfiniBand per-

```
# Automatically generated isolation config for opensm.
# Virtual clusters in OpenNebula are merged with topology
# information to form InfiniBand topologies.
#
# partition for virtual cluster vc1
# comprising VMs and hosts
# [vm-id] vm name : [host-id] host name
# [256] compnde1 : [2] hpblade01
# [255] headnode : [3] hpblade02
part1b=0xdced, ipoib, defmember=full:  0x2c9030007efba, [...] ;
#
# partition for virtual cluster vc2
# comprising VMs and hosts
# [vm-id] vm name : [host-id] host name
# [257] headnde2 : [1] hpblade03
part1c=0xb0c3, ipoib, defmember=full:  0x2c903000f3756, [...] ;
# end
```

Listing 4.3: Automatically generated partition configuration.

formance benchmark bundled with the OpenFabrics Enterprise Distribution and a small MPI job: Communication via InfiniBand has only been possible between virtual nodes in the same virtual cluster, as we configured in OpenNebula. Thus, we have successfully combined cloud management and InfiniBand network management to automatically configure network isolation for virtual clusters.

4.4.2 Scalability of OpenNebula

Up to now, we have tacitly assumed that OpenNebula can handle several thousand hosts. Clearly, operation of a productive HPC cloud requires to manage entities such as hosts and VMs in that order of magnitude. In this section, we show that OpenNebula can indeed handle these amounts.

We test how the OpenNebula management core behaves with large numbers of managed entities (hosts and VMs). We do not require a test cluster of the actual size, because we put an emulation layer below OpenNebula that only mimics physical hosts (of a configurable amount). With this setup, we discuss two potential bottlenecks, the database backend that stores information about managed entities and the command line frontends provided with OpenNebula. In both areas, we show that a straight-forward prototypic setup has serious performance problems, which, however, can be mitigated by reverting to better alternatives.

We have run all experiments mentioned in this section on our OpenNebula development system, not on the test cluster. The difference in CPU speed[8] is no obstacle for comparison, because the observed bottlenecks are much more pronounced.

[8] Our development system is equipped with a dual-core Intel Core 2 Duo T7500 CPU, clocked at 2.2 GHz, and 3 GB of RAM. In contrast, the nodes of the test cluster contain Intel Xeon E5520 quad-core CPUs, clocked at 2.26 GHz, and 6 GB of main memory.

The structure of this section reflects this introduction: First, we present the physical infrastructure emulator we have developed. Second and third, we discuss the potential bottlenecks in the form of the DB backend and the CLI tools provided with OpenNebula.

Physical Infrastructure Emulation

We employed a cluster of four nodes for our evaluation. This comparably small number of nodes does not allow to assess how OpenNebula behaves with several hundreds or several thousands of hosts. Further, the physical cluster has not always been available for tests and debugging during development. Thus, we required a way to observe OpenNebula as it interacts with a much greater number of hosts than four, and to run, test, and debug OpenNebula without using physical hosts at all. For this purpose, we developed an infrastructure emulator that only mimics physical hosts and their resources towards OpenNebula and thereby provides (emulated) clusters of almost arbitrary size. Instead of actually running VMs, our emulator only tracks their static resource allocation.

The architecture of OpenNebula uses a layer of plug-ins for operations such as querying information about a host or deploying a VM on a host. With this approach, the same management core logic can control different hypervisors (e.g., KVM, Xen, and VMware) and leave the handling of their peculiarities to plug-ins. These plug-ins are called management drivers and they are typically implemented in the Ruby scripting language [29].

To implement our emulation of physical infrastructure, we attached a new set of management drivers to this plug-in interface. Our scripts track information about the emulated hosts in a set of files and, in contrast to the existing plug-ins, do not forward the requests of the OpenNebula core to a virtualization layer. Deploying a VM on a host, and thereby allocating CPU and memory, is reflected in a modification of the file representing the respective host. Monitoring a host results in reading this file, and replying an excerpt of its contents to OpenNebula. The infrastructure emulation comprises the InfiniBand topology management that we added to OpenNebula, as introduced in Section 3.4 on page 46 and Section 4.1.4 on page 56. Each emulated host may (virtually) contain one or more HCAs.

We complement the emulated infrastructure management drivers with a script to automatically generate emulated hosts. It assigns unique host names (numbered consecutively) and InfiniBand unique ids (GUIDs), creates the files representing the emulated hosts, and announces the emulated hosts to OpenNebula.

Potential Bottleneck: Database Backend

OpenNebula stores information about all managed entities (e.g., hosts, VMs, and disk images) in a relational database. Many administrative operations issued to OpenNebula are basically simple operations in this database — for example, creating a VM (as a managed entity, not deployed on a host) or adding a host to the set of managed physical hosts. Thus, throughput and latency of database operations form the basis of the performance of OpenNebula's management operations, especially in the context of virtual clusters: When many VMs are created and modified collectively, the respective database operations occur in bursts.

OpenNebula can employ two database management systems: MySQL and SQLite. Example configurations employ a SQLite backend. It is embedded in OpenNebula as a library and has the advantage of being very easy to setup. As an alternative, MySQL runs as a separate database server and promises increased performance at the expense of more configuration work (mainly setup and maintenance of the database server). We compared the two alternatives with regard to the resulting performance of elementary management operations in OpenNebula and found MySQL to be clearly the preferable database backend for productive installations (where the slightly higher configuration work is easily amortized).

We identified an explicit performance problem in the SQLite database backend, as it is currently used in OpenNebula. SQLite by default creates and deletes a journal file in the progress of each single database transaction. With the filesystems used in our development system and in the test cluster nodes (ext3 and reiserfs), this file creation and deletion causes several disk I/O operations. In OpenNebula, a database transaction, and thus disk I/O, occurs for every single operation on every managed entity. As a result, creating many VMs, to instantiate a virtual cluster, would take an unacceptable long amount of time just to create database entries for each VM. Performance can be improved by changing the behavior of SQLite to keep the journal file between transactions[9]. We have modified OpenNebula to issue an appropriate command after opening the SQLite database. Of course, using the MySQL database management system is another alternative.

We measured the performance of an elementary management operation in OpenNebula that results from the three options SQLite, SQLite with changed journal handling, and MySQL. For this purpose, we chose the creation of a host as a managed entity in OpenNebula, because it does in fact only create an entry in the database. See Table 4.11 for the results. Performance with SQLite is almost an order of magnitude worse than with MySQL (especially with the expensive journal file operations). Hence, we clearly prefer the MySQL database backend for productive use and recommend to use SQLite only in small test installations.

	SQLite	SQLite (keep journal file)	MySQL
time for adding a host as managed entity	65 ms	14 ms	< 2 ms

Table 4.11: Time required for an elementary management operation in OpenNebula with varying DB backends.

Potential Bottleneck: Provided CLI Tools

OpenNebula exports all of its abstractions and operations via a web service interface (using the XML-RPC protocol). It provides a basic set of command line tools, written in the Ruby scripting language, that employ this interface to trigger operations such as creating a VM. Although these tools serve quite well to manually control OpenNebula in a small test environment, they are not suitable for integration into an automation framework, because they incur too much overhead on every single operation. A much better alternative for calling OpenNebula operations from other tools (e.g., a separate frontend that provides a virtual cluster interface) is to

[9] We change the behavior of SQLite with the SQLite-specific SQL statement *PRAGMA journal_mode=PERSIST;*

directly employ the web service interface. In this section, we point out the basic problem of the command line tools and compare them to the direct use of the OpenNebula web service interface with some measurements.

Each single OpenNebula operation, called via the command line tools, triggers a start of the Ruby interpreter as a new process. This Ruby interpreter initializes itself, loads the XML-RPC client libraries and the frontend library that maps the OpenNebula abstractions to web service calls, and finally executes the actual script that comprises the command line tool. Consequently, more time is spent on initialization than on actual interaction with OpenNebula.

We compare the command line tools with calling the OpenNebula web service interfaces directly for two operations: we create hosts and VMs as managed entities in OpenNebula. For this purpose, we employ the provided Java classes that map abstractions, such as VMs, to OpenNebula's web service interface in a custom test application. See Table 4.12 for the time demand per operation we have observed.

These measurements clearly indicate that using the command line tools is very inefficient and their use should consequently be restricted to manual administration tasks. In contrast, we have successfully created a thousand VMs in one short burst using our test program and OpenNebula's XML-RPC interface.

Table 4.12: Time demand of OpenNebula operations when called via provided CLI tools or directly via web service interface.

operation	time	
	CLI tools	XML-RPC directly
create a VM	160 ms	5.5 ms
add a host	100 ms	2.6 ms

Summary

We have examined elementary operations of OpenNebula. These operations are purely administrative and do not modify the actual virtual environment (such as starting VMs), so they should take minimal time to avoid unnecessary overhead on actual VM management operations. For the purpose of evaluating the core of OpenNebula itself, not an actual virtual environment, we have developed an infrastructure emulation that mimics physical hosts. Using this emulation, we were able to observe OpenNebula with a number of managed hosts and VMs far beyond the capacity of our physical test cluster. We have identified two potential bottlenecks: Employing the SQLite database, as in many example configurations, leads to disappointing performance, and the command line tools provided with OpenNebula are not feasible for automated use, because they suffer too much overhead from invoking a ruby interpreter. Both bottlenecks can be avoided however: OpenNebula can use the MySQL database that provides latencies of a few milliseconds, and the operations of OpenNebula can be triggered by other applications with low overhead via an XML-RPC interface.

5 Conclusion

An Infrastructure-as-a-Service (IaaS) model for HPC computing — that is, the concepts of cloud computing transferred to HPC clusters — promises cost savings and increased flexibility. It enables the progress away from physically owned HPC clusters to virtualized and elastic HPC resources leased from a consolidated large HPC computing center. Elastic virtual clusters provide compute capacity scaled dynamically to suit actual demand and workload. At the same time, the pay-as-you-go principle of cloud computing avoids the huge initial investments that are inevitable with physically owned clusters. It causes costs only for actual use of computing capacity. Virtualization allows to dynamically customize the complete runtime environment, from the OS kernel up to libraries and tools.

At the same time, however, virtualization raises difficult challenges. It incurs overhead and, even more important, may lead to unpredictable variations in performance [63, 93], which can severely reduce the performance of parallel applications [27, 67, 89]. Therefore, we have followed the goal to incur minimal OS background activity.

In addition, compute clusters often employ distinct cluster interconnect networks because of their performance characteristics and advanced features, which the virtualized Ethernet of general-purpose clouds fails to provide. So, we have faced the challenge to incorporate the management of cluster interconnects in cloud management. Further, multi-tenancy in HPC clouds requires network isolation, and network resources must be shared in a way that fulfills the quality of service (QoS) requirements of HPC applications.

In this work, we have presented a novel architecture for HPC clouds that provide virtual and elastic HPC clusters. Our architecture comprises the three facets node virtualization, network virtualization, and HPC cloud management. In this section, we summarize the main features of our approach and conclude the results of our evaluation.

We addressed the issue of whether a commodity hypervisor, the kernel-based virtual machine (KVM) on Linux, can be transformed to provide *virtual nodes*, virtual machines (VMs) intended for HPC workloads, that are appropriate for our architecture (in Section 3.2 on page 20). We explored the state of the art in virtualized InfiniBand access and found vendor-supported SR-IOV drivers the best choice (these drivers will be released a few weeks after the finish of this work).

We have examined the influence of various Linux kernel configurations on OS noise and application performance in virtual environments. We found that the default configuration, with a reduced timer tick frequency of 100 Hz results in an acceptable virtualization overhead of typically less than 4 % (with a maximum of 10 % only in a single benchmark configuration). Disabling swapping support can provide a slight advantage. A strict reduction of the features compiled into the Linux kernel to a minimum has shown adverse effects on performance, however. Using such a kernel as the host OS has led to significantly higher virtualization overheads

than with the other examined configurations (in some configurations by almost an order of magnitude).

Regarding OS noise, the examined configurations miss the requirements commonly posed by literature or stated in an exemplary request for proposals (RFP) for a supercomputer. Tsafrir and associates state that OS noise should not exceed the order of 10^{-5} to avoid application performance degradations in clusters with thousands of nodes [89]. The ASC Sequoia RFP [9] demands for a scaled OS noise of $< 10^{-6}$. Yet, our results indicate scaled noise means in the range of 10^{-4} to 10^{-2}. These results show, that Linux is no preferable choice for large clusters with thousands of nodes. Based on our measurements of virtualization overhead, however, we can nevertheless conclude that KVM and Linux present an appropriate virtualization layer with support for InfiniBand virtualization for small- to medium-sized HPC workloads (most real-world parallel compute jobs employ only up to 128 processes [40], anyway).

We suggest to dedicate physical CPU cores to logical CPU cores for HPC workload. We have shown by experimentation that the alternative policy, timesharing, can cause disproportionately severe reductions of the performance of HPC applications. Therefore, omitting consolidation of several logical CPUs, but dedicating physical CPU cores is a reasonable choice. As an alternative, we propose to run low-priority VMs with non-HPC background workloads to increase CPU utilization. We found that existing Linux scheduling policies can be employed for this purpose: In our experiments, fully loaded background VMs caused only minimal performance degradations for CPU-intensive HPC applications. Memory-intensive workloads, however, can be impaired by low-priority background VMs running on different CPU cores. This effect demonstrates that OS scheduling policies have no influence on the arbitration of hardware resources, such as shared memory buses.

Concerning the second aspect of our architecture, we provided an extensive concept for InfiniBand network virtualization that provides each user with the impression of exclusive access to a dedicated physical network. A user can apply a custom routing scheme, employ recursive isolation in his share of the network, and assign custom node addresses (compare Section 3.3.1 on page 31). However, his configuration settings affect only his share of the network and cannot impair other users. We have presented an in-depth approach on how we provide this virtual network view to VMs using SR-IOV backend drivers.

We have run experiments on the enforcement of network isolation on actual InfiniBand network hardware. We checked the practical availability of the feature that protects the configuration of an InfiniBand network against illegitimate modifications (using protection keys for authorization). We found that current versions of the *opensm* subnet manager, a commonly used InfiniBand network management tool, fails to configure the protection mechanism correctly. After identifying and fixing this problem, we verified the proper function of the protection mechanism with several InfiniBand diagnostics tools that try to both modify or just query several configuration settings. With our correction in place, the configuration of an InfiniBand network, thus network and performance isolation, cannot be altered or even retrieved by users of the HPC cloud, even if they has complete access to an InfiniBand host channel adapter (virtualization adds an additional protection layer).

We have discussed the additional challenges that cluster networks bring up for the third facet of our architecture — HPC cloud management. The peculiarities of cluster networks, such

as the QoS mechanisms of InfiniBand, and performance requirements place constraints on the distribution of virtual clusters in the physical network. To face these challenges, we introduced extensions to existing cloud management frameworks: We incorporated network topology information into cloud management, so that network distance and the restrictions incurred by the InfiniBand QoS mechanisms can be considered in VM placement.

As a proof of concept of our approach, we have developed a prototype that combines cloud and cluster network management. Our prototype queries identifiers of the InfiniBand host channel adapters of physical hosts on the one hand, and allows to group VMs into virtual clusters on the other hand. Then, it combines both aspects and automatically generates a configuration of InfiniBand network isolation between the defined virtual clusters. Despite the limitations of our test cluster (no SR-IOV support), we have run a scenario with two distinct virtual clusters and verified successfully that the automatically configured network isolation matches the defined virtual clusters.

5.1 Future work

In this work, we have presented a novel architecture for HPC clouds and discussed its three facets node virtualization, network virtualization, and HPC cloud management. We have evaluated building blocks of this architecture and developed a prototype for automatic configuration of network isolation. In this section, we conclude this work by pointing out some directly related topics of future work.

Our observation of increased virtualization overhead with a minimal Linux kernel configuration contradicts our intuitive guess that this kernel should provide the best performance. In further studies, we will analyze the influence of different kernel configurations in greater detail to determine the cause of the additional overhead.

We are currently trying to get SR-IOV support working in our test cluster (first tests failed because of the BIOS compatibility issues we discussed). With this virtualization technique running, we will extend our current prototype to include transparent network virtualization for VMs, based on SR-IOV, as introduced in Section 3.3.1 on page 31. It will allow us to proof our claim, that we can achieve such virtualization without overhead on network bandwidth or latency.

This work leaves open the question, which QoS levels an HPC cloud can guarantee to a virtual cluster. Early experiments indicate that latency, as experienced by a virtual cluster, can be bound even on completely utilized network links. However, these experiments are not yet finished and therefore have not been included in this work. In addition, complex network topologies raise the question how to derive performance guarantees for large, real-life networks from measurements in small test networks (comprising a single switch) — literature already provides performance models for InfiniBand networks, and we expect that they can be employed for this purpose.

Our current prototype is based on the OpenNebula cloud management framework [65, 84]. However, we will evaluate the nimbus toolkit [47, 70] as a potential alternative. Both frameworks have a similar architecture that is designed with extensibility in mind. However, the

nimbus toolkit has built-in support for virtual clusters and provides a more flexible contextualization mechanism.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 2–13. ACM, 2006.

[2] *AMD64 Architecture Programmer's Manual*, volume 2: System Programming. Advanced Micro Devices, Inc., 2010.

[3] *AMD I/O Virtualization Technology (IOMMU) Specification*. Advanced Micro Devices, Inc., February 2009.

[4] Paolo Anedda, Simone Leo, Simone Manca, Massimo Gaggero, and Gianluigi Zanetti. Suspending, migrating and resuming hpc virtual clusters. *Future Generation Comp. Syst.*, 26(8):1063–1072, 2010.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and David E. Culler. Effective distributed scheduling of parallel workloads. In *SIGMETRICS*, pages 25–36, 1996.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[8] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[9] Asc sequoia request for proposals. http://asc.llnl.gov/sequoia/rfp/.

[10] Asc sequoia benchmark codes. http://asc.llnl.gov/sequoia/benchmarks/.

[11] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, pages 45–58, 1999.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 164–177. ACM, 2003.

[13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[14] Theophilus Benson, Aditya Akella, Sambit Sahu, and Anees Shaikh. Epic: Platform-as-a-service model for cloud networking. Technical Report 1686, CS Department, University of Wisconsin, Madison, February 2011.

[15] Leonid B. Boguslavsky, Karim Harzallah, Alexander Y. Kreinin, Kenneth C. Sevcik, and Alexander Vainshtein. Optimal strategies for spinning and blocking. *J. Parallel Distrib. Comput.*, 21(2):246–254, 1994.

[16] Rajkumar Buyya. Market-oriented cloud computing: Vision, hype, and reality of delivering computing as the 5th utility. In Franck Cappello, Cho-Li Wang, and Rajkumar Buyya, editors, *CCGRID*, page 1. IEEE Computer Society, 2009.

[17] Fernando L. Camargos, Gabriel Girard, and Benoit D. Ligneris. Virtualization of Linux servers: a comparative study. In *Ottawa Linux Symposium*, pages 63–76, July 2008.

[18] Ludmila Cherkasova and Rob Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 387–390. USENIX, 2005.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI*. USENIX, 2005.

[20] Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[21] J. M. Dennis and H. M. Tufo. Scaling climate simulation applications on the ibm blue gene/l system. *IBM J. Res. Dev.*, 52:117–126, January 2008.

[22] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. In *HPCA*, pages 1–10. IEEE Computer Society, 2010.

[23] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In Muli Ben-Yehuda, Alan L. Cox, and Scott Rixner, editors, *Workshop on I/O Virtualization*. USENIX Association, 2008.

[24] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[25] Tarek A. El-Ghazawi, Kris Gaj, Nikitas A. Alexandridis, Frederic Vroman, Nguyen Nguyen, Jacek R. Radzikowski, Preeyapong Samipagdi, and Suboh A. Suboh. A performance study of job management systems. *Concurrency - Practice and Experience*, 16(13):1229–1246, 2004.

[26] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - a status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004.

[27] Kurt B. Ferreira, Patrick G. Bridges, and Ron Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *SC*, page 19. IEEE/ACM, 2008.

[28] Renato J. O. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A case for grid computing on virtual machine. In *ICDCS*, pages 550–559. IEEE Computer Society, 2003.

[29] David Flanagan and Yukihiro Matsumoto. *The Ruby programming language - everything you need to know: covers Ruby 1.8 and 1.9*. O'Reilly, 2008.

[30] Message Passing Interface Forum. Mpi: A message-passing interface standard, 1994.

[31] Ian T. Foster, Timothy Freeman, Katarzyna Keahey, Doug Scheftner, Borja Sotomayor, and Xuehai Zhang. Virtual clusters for grid communities. In *CCGRID*, pages 513–520. IEEE Computer Society, 2006.

[32] Abel Gordon, Michael Hines, Dilma Da Silva, Muli Ben-Yehuda, Marcio Silva, and Gabriel Lizarraga. Ginkgo: Automated, Application-Driven Memory Overcommitment for Cloud Computing. In *ASPLOS RESoLVE '11: Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE) workshop*, 2011.

[33] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open mpi: A flexible high performance mpi. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors, *PPAM*, volume 3911 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 2005.

[34] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In Pablo Rodriguez, Ernst W. Biersack, Konstantina Papagiannaki, and Luigi Rizzo, editors, *SIGCOMM*, pages 51–62. ACM, 2009.

[35] Peter H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.

[36] Wei Huang. *High Performance Network I/O in Virtual Machines over Modern Interconnects*. PhD thesis, Ohio State University, 2008.

[37] Wei Huang, Jiuxing Liu, Bülent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In Gregory K. Egan and Yoichi Muraoka, editors, *ICS*, pages 125–134. ACM, 2006.

[38] *InfiniBand Architecture Specification Volume 1, Release 1.2.1*. InfiniBand Trade Association, 2007.

[39] *Intel Virtualization Technology for Directed I/O Architecture Specification*. Intel Corp., Sept 2008.

[40] Alexandru Iosup, Catalin Dumitrescu, Dick H. J. Epema, Hui Li, and Lex Wolters. How are real grids used? the analysis of four grid traces and its implications. In *GRID*, pages 262–269. IEEE, 2006.

[41] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2011.

[42] *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[43] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010.

[44] Rebecca Isaacs and Yuanyuan Zhou, editors. *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*. USENIX Association, 2008.

[45] Terry Jones, Shawn Dawson, Rob Neely, William G. Tuel Jr., Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In sc2003 [77], page 10.

[46] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, G. Bruce Berriman, Benjamin P. Berman, and Philip Maechling. Scientific workflow applications on amazon ec2. *CoRR*, abs/1005.2718, 2010.

[47] Katarzyna Keahey and Tim Freeman. Contextualization: Providing one-click virtual clusters. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

[48] Darren J. Kerbyson and Philip W. Jones. A performance model of the parallel ocean program. *IJHPCA*, 19(3):261–276, 2005.

[49] Kernel-based virtual machine (kvm). http://www.linux-kvm.org/.

[50] John R. Lange, Kevin T. Pedretti, Trammell Hudson, Peter A. Dinda, Zheng Cui, Lei Xia, Patrick G. Bridges, Andy Gocke, Steven Jaconette, Michael Levenhagen, and Ron Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In IPDPS2010 [43], pages 1–12.

[51] libvirt. http://www.libvirt.org/.

[52] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In IPDPS2010 [43], pages 1–12.

[53] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track* [91], pages 29–42.

[54] Kieran Mansley, Greg Law, David Riddoch, Guido Barzini, Neil Turton, and Steven Pope. Getting 10 gb/s from xen: Safe and fast device access from unprivileged domains. In Luc Bougé, Martti Forsell, Jesper Larsson Träff, Achim Streit, Wolfgang Ziegler, Michael Alexander, and Stephen Childs, editors, *Euro-Par Workshops*, volume 4854 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2007.

[55] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[56] Paul Marshall, Kate Keahey, and Timothy Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGRID*, pages 43–52. IEEE, 2010.

[57] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[58] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in xen (awarded best paper). In *USENIX Annual Technical Conference, General Track* [91], pages 15–28.

[59] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *Operating Systems Review*, 40(2):8–11, 2006.

[60] Hans Werner Meuer. The top500 project: Looking back over 15 years of supercomputing experience. *Informatik Spektrum*, 31(3):203–222, 2008.

[61] Rubén S. Montero, Rafael Moreno-Vozmediano, and Ignacio Martín Llorente. An elasticity model for high throughput computing clusters. *J. Parallel Distrib. Comput.*, 71(6):750–757, 2011.

[62] Anastassios Nanos and Nectarios Koziris. Myrixen: Message passing in xen virtual machines over myrinet and ethernet. In Hai-Xiang Lin, Michael Alexander, Martti Forsell, Andreas Knüpfer, Radu Prodan, Leonel Sousa, and Achim Streit, editors, *Euro-Par Workshops*, volume 6043 of *Lecture Notes in Computer Science*, pages 395–403. Springer, 2009.

[63] J. Napper and P. Bientinesi. Can cloud computing reach the top500? In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 17–20. ACM, 2009.

[64] Openfabrics alliance. http://www.openfabrics.org/.

[65] Opennebula - the open source toolkit for cloud computing. http://opennebula.org/.

[66] John K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS*, pages 22–30. IEEE Computer Society, 1982.

[67] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In sc2003 [77], page 55.

[68] Parallel ocean program (pop). http://climate.lanl.gov/Models/POP/.

[69] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[70] Nimbus project. Nimbus toolkit website. http://www.nimbusproject.org.

[71] Qemu. http://www.qemu.org/.

[72] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In Carl Kesselman, Jack Dongarra, and David W. Walker, editors, *HPDC*, pages 179–188. ACM, 2007.

[73] A. Ranadive, A. Gavrilovska, and K. Schwan. FaRes: Fair resource scheduling for vmm-bypass infiniband devices. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 418–427. IEEE Computer Society, 2010.

[74] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040 (Proposed Standard), October 2007.

[75] John S. Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, page 10, Berkeley, CA, USA, 2000. USENIX Association.

[76] Jose Renato Santos, Yoshio Turner, G. John Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In Isaacs and Zhou [44], pages 29–42.

[77] *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom*. ACM, 2003.

[78] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN*, pages 249–258. IEEE Computer Society, 2006.

[79] Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, Willy Zwaenepoel, and Paul Willmann. Concurrent direct network access for virtual machine monitors. In *HPCA*, pages 306–317. IEEE Computer Society, 2007.

[80] Tom Shanley. *InfiniBand Network Architecture*. MindShare System Architecture. Addison-Wesley Professional, Boston, MA, USA, 2002.

[81] Galen M. Shipman, Timothy S. Woodall, Richard L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. Infiniband scalability in open mpi. In IPDPS2006 [42].

[82] *Single Root I/O Virtualization and Sharing 1.1 Specification*. PCI Special Interest Group, 2010.

[83] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer*, 38(5):32–38, 2005.

[84] Borja Sotomayor, Rubén S. Montero, Ignacio Martín Llorente, and Ian T. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[85] Matthew J. Sottile and Ronald Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *CLUSTER*, pages 371–377. IEEE Computer Society, 2004.

[86] Gary Stiehr and Roger D. Chamberlain. Improving cluster utilization through intelligent processor sharing. In IPDPS2006 [42].

[87] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In Yoonho Park, editor, *USENIX Annual Technical Conference, General Track*, pages 1–14. USENIX, 2001.

[88] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[89] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In Arvind and Larry Rudolph, editors, *ICS*, pages 303–312. ACM, 2005.

[90] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *IEEE Computer*, 38(5):48–56, 2005.

[91] USENIX. *Proceedings of the 2006 USENIX Annual Technical Conference, May 30 - June 3, 2006, Boston, MA, USA*. USENIX, 2006.

[92] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[93] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10. IEEE, 2010.

[94] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *OSDI*, 2002.

[95] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized i/o devices. In Isaacs and Zhou [44], pages 15–28.

[96] Chris Wright. Pci direct device assignment: pwned! all your device are belong to guest. In *KVM Forum 2010*, 2010. `http://www.linux-kvm.org/page/KVM_Forum_2010`, retrieved on Apr 05, 2011.

[97] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, IBM Research, 2008.

[98] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2003.