Flexible Device Sharing in PCIe Clusters using Device Lending

Jonas Markussen*
Simula Research Laboratory
Oslo, Norway
jonassm@simula.no

Lars Bjørlykke Kristiansen
Dolphin Interconnect Solution AS
Oslo, Norway
larsk@dolphinics.no

Håkon Kvale Stensland*
Simula Research Laboratory
Oslo, Norway
haakonks@simula.no

Friedrich Seifert
Dolphin Interconnect Solution AS
Oslo, Norway

Carsten Griwodz†
University of Oslo
Oslo, Norway

Pål Halvorsen*
Simula Research Laboratory
Oslo, Norway

ABSTRACT

Processing workloads may have very high IO demands, exceeding the capabilities provided by resource virtualization and requiring direct access to the physical hardware. For computers that are interconnected in PCI Express (PCIe) networks, we have previously proposed Device Lending as a solution for assigning devices to remote hosts. In this paper, we explain how we have extended our implementation with support for the Linux Kernel-based Virtual Machine (KVM) hypervisor. Using our extended Device Lending, it becomes possible to dynamically "pass through" physical remote devices to VM guests while still retaining the flexibility of virtualization, something that previously required extensive facilitation in both hypervisor and device drivers in the form of paravirtualization.

We have also improved our original implementation with support for interoperability between remote devices. We show that it is possible to use multiple devices residing in different hosts, while still achieving the same bandwidth and latency as native PCIe, and without requiring any additional support in device drivers.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; *Interconnection architectures*; Cloud computing;

KEYWORDS

Resource sharing, resource allocation, networked resources, virtualization, PCIe, data access, IOMMU, non-transparent bridging

ACM Reference Format:
Jonas Markussen, Lars Bjørlykke Kristiansen, Håkon Kvale Stensland, Friedrich Seifert, Carsten Griwodz, and Pål Halvorsen. 2018. Flexible Device Sharing in PCIe Clusters using Device Lending . In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3229710.3229759

*Also with University of Oslo.
†Also with Simula Research Laboratory.

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229759

1 INTRODUCTION

Different processing workloads can have highly variable demands to processing power and IO resources. Cloud providers, such as Amazon AWS and Microsoft Azure, often base their pricing models on offering different, or even custom, IO device configurations for their VM images. However, as physical hardware resources may be limited, it is desirable to be able to scale up and allocate more resources and release them on demand. Dynamic scaling based on current workload requirements leads to more efficient utilization of the available physical resources.

Such scaling is made possible by VM hypervisors through resource virtualization, primarily software emulation and paravirtualization. Software-emulated devices appear to the VM guest as an IO device, but all functionality is handled in the VM implementation. Paravirtualized devices also offer device functionality in software, but the software-defined device resembles the physical device more closely. As both methods of resource virtualization require facilitation in the hypervisor, the availability of different types of resources is limited by the underlying virtualization technology being used. In addition, workloads that rely on multi-device interoperability becomes a challenge, as setting up necessary memory mappings for Remote Direct Memory Access (RDMA) and device-to-device access is generally not possible without extensive facilitation in both the hypervisor and VM guests themselves.

Many modern processors implement an IO Memory Management Unit (IOMMU), allowing devices to be *passed through* to a VM instance, without compromising the memory encapsulation provided by the virtualized environment. While pass-through allows physical hardware to be used with minimal software overhead, this technique does not have the flexibility of resource virtualization; using pass-through, VM instances become tightly coupled with the resources they use, and distributing VMs across multiple hosts in a way that maximizes utilization becomes a challenge.

For machines that are interconnected in a PCIe cluster, where IO devices and interconnection technology are attached to the same PCIe fabric, we have proposed a different strategy to resource sharing using Device Lending [15]. Device Lending exploits the memory addressing capabilities inherent in PCIe networks in order to decouple devices from the hosts they physically reside in, allowing them to be dynamically reassigned to different machines and used as if they were locally installed.

In this paper, we describe our improved Device Lending concept by extending it with support for the KVM hypervisor, allowing physical remote devices to be passed through to a VM instance.

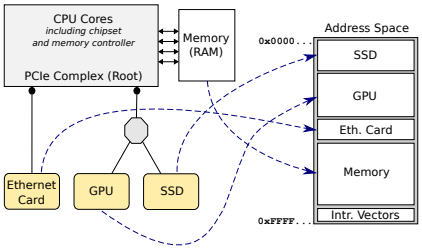


Figure 1: Device memory is mapped into the same address space as the CPUs, allowing devices to access both system memory and other devices.

We have also implemented support for direct device-to-device access, enabling true multi-device interoperability. Finally, we also investigate the impact of IO address virtualization on performance, particularly in the case of device-to-device access. Our findings show that we are able to borrow and use multiple remote devices, achieving the same bandwidth as native PCIe and without adding any additional latency beyond that of the interconnect. With virtualization support, it is possible for Cloud providers to offer highly customizable configurations of devices that are passed through to VMs. Combined with support for efficient device-to-device data transfers, it is possible to create highly flexible and dynamic configurations of local and remote IO devices in a PCIe cluster.

The remainder of this paper is organized as follows: we present essential capabilities of PCIe in Section 2. In Section 3, we discuss related work. In Section 4, we provide an outline of our original Device Lending implementation. We describe how we have extended Device Lending with virtualization support in Section 5. Section 6 describes how we have added support for borrowing from multiple lenders, followed by a performance evaluation in Section 7. A summary of our findings and conclusion is presented in Section 8.

2 PCIE OVERVIEW

PCIe is today the most widely adopted industry standard for connecting hardware peripherals (devices) to a computer system [10]. Device memory, such as register and onboard memory are mapped into an address space shared with the CPUs and their memory controllers (Figure 1). Memory operations, such as reads and writes, are transparently routed onto the PCIe fabric. This enables a CPU to access device memory, as well as allowing devices capable of DMA to directly read and write to system memory.

PCIe uses point-to-point links, where a link consists of 1 up to 16 lanes. Each lane is a full-duplex serial connection. Data is striped across multiple lanes and wider links yield higher bandwidths. The current revision, PCIe Gen3 [21], specifies a theoretical maximum data rate of 984.5 MB/s per lane.

Not unlike other networking technologies, PCIe also uses a layered protocol. The uppermost layer is called the transaction layer, and one of its responsibilities is to forward memory reads and writes as transaction layer packets (TLPs). It is also responsible for packet ordering, meaning that memory operations in PCIe are strictly ordered. Underneath the transaction layer lies the data link layer and the physical layer, and their responsibilities include flow control, error correction, and signal encoding.

As shown in Figure 2, the entire PCIe network is structured as a tree, where devices form the leaf nodes. In PCIe terminology,

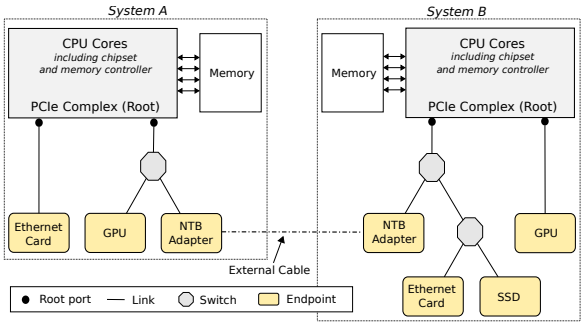


Figure 2: Example of a PCIe topology. Two independent networks are connected together using an NTB. The NTB translates IO addresses between the two different address spaces, creating a shared address space between the networks.

a device is therefore referred to as an "endpoint". Switches can be used to create subtrees in the network. The "root ports" are at the top of the tree, and act as the connection between the PCIe network and the CPU cores (CPUs, chipset, and memory controller). The entire PCIe network comprises the "fabric". Note that in the figure, two independent network roots are interconnected using a Non-Transparent Bridge (NTB), which we will explain below.

2.1 Memory addressing and forwarding

The defining feature of PCIe is that devices are mapped into the same address space as the CPU and system memory (Figure 1). Because this mapping exists, a CPU is able to read from and write to device memory regions, the same way it would read from system memory. No specialized port IO is required. Likewise, if a device is capable of DMA, it can read from and write to system memory, as well as other devices on the fabric.

In order to map device memory regions to address ranges, the system scans the PCIe tree and accesses the configuration space of each device attached to the fabric. The configuration space describes the capabilities of the device, such as describing the device's memory regions. Switches in the topology are assigned the combined address range of their downstream devices. This allows forwarding of memory operations based on address ranges to occur in a strictly hierarchical fashion in the tree, and TLPs are forwarded either upstream or downstream. An important property of this hierarchical routing is that packets do not need to pass through the root, but can be routed using the shortest path if the chipset allows it. This is referred to as peer-to-peer in PCIe terminology. Using Figure 2, System B's lower switch will have the address range of both the Ethernet card and the SSD, allowing TLPs to be routed directly between them, device to device, without passing through the root.

Another significant feature of PCIe, is the use of message-signalled interrupts (MSI) instead of physical interrupt lines. MSI-capable devices post a memory write TLP to the root using a pre-determined address. The write TLP is then interpreted by the CPU, which uses the payload to raise an interrupt specified by the device.

2.2 Virtualization support and pass-through

Modern processor architectures implement IOMMUs, such as Intel VT-d [3]. The IOMMU provides virtualization of addresses between the PCIe fabric and the CPU (including memory controllers). One of

the most important features of the IOMMU is the ability to translate addresses of DMA operations from any IO device [1]. In other words, it translates virtual IO addresses to physical addresses.

Similarly to pages mapped by an MMU for individual userspace processes, an IOMMU can group PCIe devices into IOMMU domains. As each domain has its own individual mappings, members of an IOMMU domain consequently have their own private virtual address space. Such a domain can be part of the virtualized address space of a VM, while other PCIe devices and the rest of memory remain isolated. This allows the VM to interact directly with the device using native device drivers from within the guest, while the host retains the memory isolation provided by the virtualization. This is often referred to as "pass-through".

As most device drivers make the assumption that they have exclusive control over a device, sharing a device between several VM instances requires either paravirtualization, such as Nvidia vGPUs [17], or SR-IOV [22]. SR-IOV-capable devices allow a single physical device to act as multiple virtual devices, allowing a hypervisor to map the same device to several VMs.[1]

2.3 Non-Transparent Bridging

Because of its high bandwidth and low latency, it is desirable to extend the PCIe fabric out of a single computer and use it for high-speed interconnection networks [23]. This can be accomplished using an NTB implementation [24]. Although not standardized, NTBs are a widely adopted solution for interconnecting independent PCIe network roots, and all NTB implementations have similar capabilities. Some processor architectures, such as recent Intel Xeon and AMD Zen, have a built-in NTB implementation [27].

Despite the name, an NTB actually appears as a PCIe endpoint. This is illustrated in Figure 2, where the connected systems have their own NTB adapter card. Just like regular endpoints, they appear to have one or more memory regions that can be read from or written to by CPUs or other devices. Memory operations on these regions are forwarded from one PCIe network to the other. As the interconnected networks use different layouts for their address space, the NTB performs a hardware address translation on the TLPs during the forwarding. Consequently, NTBs create a shared memory architecture between separate systems with very low additional overhead in terms of latency.

As the address ranges associated with the NTB may be too small to cover the entire address space of the different systems, some NTBs support dividing their range into segments. A segment can be mapped anywhere into the remote system's address space. Due to the complexity of translating addresses in hardware, the number of possible mappings to remote systems is limited.

3 RELATED WORK

The idea of a unified network for the inner components of a computer with those of another is not new. It was already imagined for both ATM [26] and SCI [4]. These ideas never got implemented, because none of these technologies were picked up for internal IO interconnection networks.

PCIe is the dominant standard for internal IO bus, and is also proving to be a relevant contender for external interconnection networks. PCIe, however, was designed to be used within a single computer system only. In this section, we will discuss some solutions for sharing IO devices between multiple hosts.

3.1 Distributed IO using RDMA

There are several technologies which are more widely adopted for creating high-speed interconnection networks than PCIe. These include InfiniBand, as well as 10Gb and 40Gb Ethernet [5, 16]. To make use of their high throughput, they rely on RDMA [29]. Variants are summarized by Huang et al. [12] and include native RDMA over InfiniBand, Converged Enhanced Ethernet (RoCE), and Internet Wide Area RDMA Protocol (iWARP). To alleviate the complexity of programming for RDMA, middleware extensions like RDMA for MPI-2 [14] and rCUDA [9] have been developed. Those middleware extensions have also been extended with device-specific protocols like GPUDirect for RDMA [25, 31] or NVMe over Fabrics.

While RDMA extensions may achieve very high throughput on the interconnection links, they are not as closely integrated with the IO bus fabric as PCIe, and require translation between protocol stacks. Another drawback is that it is currently only possible for such protocols to work with devices and device drivers that explicitly supports them. A proposed approach for overcoming the protocol translation overhead would be to integrate network interface functionality directly into SoCs [7], but the improvement only takes effect when the SoCs are in communication with each other. This idea is followed in the rack-scale architecture [6], which generalizes a trend returning from switched cluster architectures to hypercube architectures [11, 32]. These approaches all focus on efficient data exchange for parallel processing, rather than on resource sharing between logically separate compute units.

3.2 Virtualization approaches

Multi-Root IO Virtualization (MR-IOV) [19] specifies how several hosts can be connected to the same PCIe fabric. The fabric is logically partitioned into separate virtual hierarchies, i.e., PCIe roots, where each host sees its own hierarchy without knowing about MR-IOV. MR-IOV requires multi-root aware PCIe switches, and, in the same way as SR-IOV requires SR-IOV-aware devices to be able to provide virtual devices to several VMs, devices must be multi-root aware to provide virtual devices to several PCIe roots (and thus hosts) at the same time.

Despite being standardized in 2008 [19], we are not aware of any MR-IOV-capable devices. Instead, there are attempts to achieve MR-IOV-like functionality through a combination of SR-IOV with NTB-like hardware [28].[2]

Another virtualization approach is the Landon system [30]. Landon uses all PCIe and virtualization features as proposed in this paper, but it achieves less freedom than our Device Lending as devices are physically installed in a dedicated management host that is able distribute devices to different remote guest VMs. In addition, devices are assigned for the lifetime of the guest OS, and can not be easily reassigned on the fly.

[1] Note that Device Lending does not make any distinction between physical devices and SR-IOV virtual devices.

[2] This is also possible with Device Lending, see footnote 1.

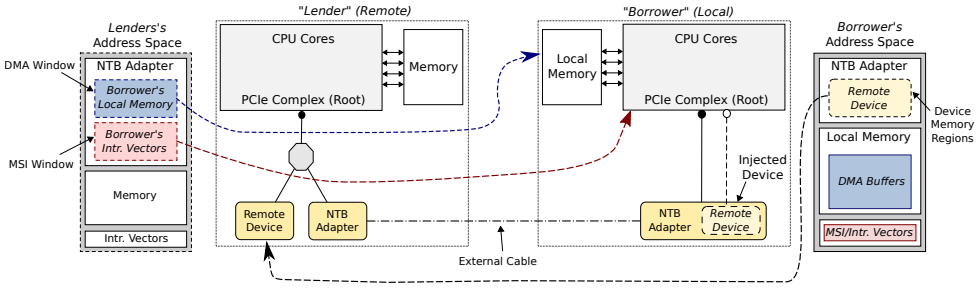


Figure 3: Using an NTB, it is possible to map the memory regions of a remote device so local CPUs are able to read and write to device registers. The remote system can in turn reverse-map the local system's memory and CPUs for the device, making DMA and MSI possible. Device Lending injects a hot-added device into the Linux kernel device tree using these mappings.

3.3 Partitioning the fabric

Rack-scale computers are so-called converged infrastructure systems, where both IO devices and interconnects are attached to a shared PCIe fabric. Rack-scale relies on dynamically partitioning the shared fabric into different subfabrics (using fabric IDs), in order to assign individual devices to different CPUs. Unlike MR-IOV, rack-scale does not require support in devices, but it does require dedicated hardware switches which support the fabric ID header extension in order to configure routes between devices and CPUs. Additionally, these systems are only modular to the extent of typical blade server configurations, and scaling beyond a single system requires facilitation using traditional distributed methods. Adding new IO devices requires additional modules, often only available from the same vendor.

There have been some efforts in achieving live-partitioning using PLX PCIe switches [33], but a performance evaluation of this appears to be lacking.

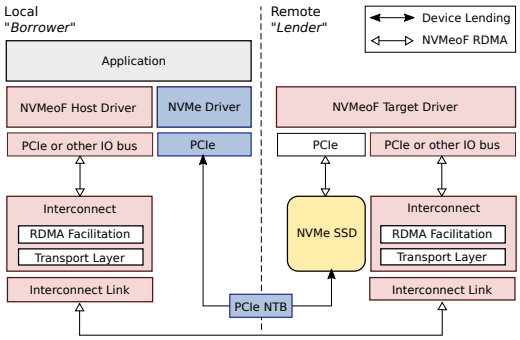


Figure 4: Illustration of native NVMe using Device Lending compared to NVMe over Fabrics using RDMA. Device Lending makes remote devices appear as if they are locally installed and there is no need for specialized support in devices or drivers.

4 DEVICE LENDING

As illustrated in Figure 3, it is possible to map the memory regions of remote PCIe devices using an NTB. A local CPU can perform memory operations on a remote device, such as reading from or writing to registers. Conversely, it is also possible to map local resources for the remote device, allowing it to write MSI interrupts and access the local system's memory across the NTB.

In order to make such mappings transparent to both devices and their drivers, we have previously implemented Device Lending [15] for an unmodified Linux kernel. Our implementation is composed of two parts, namely a "lender", allowing a remote unit to use its device, and the "borrower" using the device. By emulating a hot-plug event [23] while the system is running, we insert a virtual device into the borrower's local device tree, making it appear to the system and device driver as if a device was hot-added in the system. The device's memory regions are mapped through the NTB, allowing the local driver to read and write to device registers without being aware that the device is actually remote.

The lender is responsible for setting up reverse mappings for DMA and MSI. [3] As mentioned in Section 2.3, the address range of the NTB is not necessarily large enough to cover the entire address space of the borrowing system. Since it is generally not possible to know in advance which memory addresses a device driver might use for DMA transfers, we use an IOMMU on the borrower to set up dynamic mappings to arbitrary addresses, allowing the lender to set up a single DMA window. When the device driver calls the Linux DMA API in order to create DMA buffers, the borrower intercepts these calls. The borrower injects the IO address of the DMA window prepared by the lender and sets up a local IOMMU mapping to the DMA buffer. The driver then passes the injected address to the device, completely unaware that the address is actually a far-side address. This allows the device to reach across the NTB, transparent to both driver and device. All address translations between the different address domains are done in hardware (NTB and IOMMU), meaning that we achieve native PCIe performance in the data path.

By allowing remote devices to appear to a system as if they are locally installed, Device Lending is a method for decoupling devices from the systems they physically reside in. As hosts can act as both lender and borrower, we have created a highly flexible method of assigning and reassigning devices to computers that currently need them. We imagine this as hosts in the cluster contributing to a pool of IO resources that can be cooperatively time-shared among them. This has advantages over distributed IO using traditional approaches; network interfaces can be assigned to a computer while it needs high throughput, and released when it is no longer needed;

[3] Legacy interrupts are not supported in the current Device Lending implementation, as they can not be remapped over the NTB.

access latency in NVMe over Fabrics using RDMA can be eliminated by borrowing the NVMe disk instead and accessing it directly, as shown in Figure 4; large-scale CUDA programming tasks can make use of multiple GPUs that appear to be local instead of relying on middleware such as rCUDA [9]. In contrast to RDMA solutions, Device Lending works for *all* PCIe devices, and do not require any additional support in drivers.

Our original implementation, however, did not account for device-to-device access when borrowing multiple devices from different lenders. As the borrowing system is not aware that the devices reside in different systems, we need a mechanism to resolve IO addresses to other borrowed devices, in order to fully achieve device interoperability. In addition, our original implementation lacked support for borrowers that are VM guests. Adding virtualization support would greatly increase the usability of Device Lending, as we introduce the flexibility of decoupled remote devices and be able to dynamically assign devices using pass-through.

5 SUPPORTING VIRTUAL BORROWERS

Many modern architectures now implement IOMMUs, allowing DMA and interrupts to be remapped. This makes it possible for a driver running in a VM guest to access a device directly without breaking out of the memory isolation, as the driver is able to communicate with the device using IO virtual addresses. In Linux, such pass-through of devices is supported in the KVM hypervisor using the Virtual Function IO API [2] (VFIO). This API provides a set of functions for mapping memory for the device and control functionality, such as resetting the device, that the hypervisor can call in order to set up necessary mappings for a VM instance.

A theoretical solution for passing through remote devices, would be for the physical host to borrow the remote device, injecting the device into its local device tree, and then implement these functions. Such a solution would not be feasible due to the following reasons:

(1) The device would be borrowed by the physical host for as long as it runs, regardless of whether any VM instances would currently be using it or not. This would lead to poor utilization of device resources.
(2) All devices borrowed by the same physical host would be placed in to the same IOMMU domain by Device Lending. KVM requires pass-through devices to be placed in a separate IOMMU domain in order to prevent memory accesses that could potentially break out of the memory isolation provided by virtualization.
(3) Pass-through requires the entire address space of the guest VM to be mapped for the device. As there is no method of establishing this mapping before the VM instance is running, we need a mechanism for pinning memory pages used by the instance in order to create a DMA window.

In the 4.10 version of the Linux kernel, an extension to the VFIO API called Mediated Devices (mdev) [13] was included. This extension makes it possible to use VFIO for *paravirtualized* devices. It introduces the concept of a physical parent device having virtual child devices. This allows mdev to intercept certain operations, such as when the VM instance tries to access the device's configuration space, or when KVM is setting up interrupts. The idea is that a single physical device can be used to emulate multiple virtual devices. In our case, using the mdev extension provides us with finer grained control over what the hypervisor and guest OS is attempting to do with the device than with the "plain" VFIO API.

Our prototype creates an mdev child device when a device is discovered. This allows a hypervisor to pass through the device to a VM instance without it being borrowed (and locally injected). When the guest OS boots up and attempts to reset the device, we do the actual borrowing. When the guest OS releases the device, either by shutting down or because the VM instance hot-removes it, we return the device. Not only does this solve the issue with the lifetime of a borrowed device mentioned in (1), but it also makes it possible to hot-add a device to a live VM instance.

As we now have control over when a device is being used, and which VM instance is using it, resolving (2) becomes a matter of setting up appropriate IOMMU groups. The borrower places the mdev child device in an IOMMU group that satisfies isolation requirements by KVM. In addition, when the device is borrowed, we establish an IOMMU domain on the *lender*-side as well, in order to map the future DMA window as well as protecting against rogue memory accesses.

While other implementations using mdev implement virtual child devices, each with their own set of *emulated* resources, we are passing through the *physical device itself*. This difference becomes apparent when the guest driver initiates DMA transfers; virtual device implementations emulate device registers, and are therefore able to notify KVM to pin the appropriate memory pages before initiating the physical DMA engine. In our case, the VM instance maps the physical device registers and accesses the device directly, which means that without making assumptions about the type of device being used and implementing virtual registers for it, we are not able to replicate this specific behavior. As mentioned in (3), we are also not able to make KVM pin any memory pages until the VM instance is actually loaded and the guest OS boots up, because only then will the memory used by the VM actually be allocated.

However, in order for a device to do DMA, a dedicated register in the device's configuration space must be set. This register is common for all PCIe devices. Relying on the assumption that this register is disabled until the guest OS is booting up (and memory for the instance has been allocated), our solution is to intercept when a configuration cycle enables this register, and then notify KVM to pin pages. With the pages now locked in memory, we are able to properly set up a DMA window to memory used by the VM instance using the lender-side IOMMU domain we prepared earlier.

Finally, VFIO and mdev use the eventfd API to trigger interrupts in the VM instance. Our current prototype intercepts calls to the configuration space that enables interrupts and sets up an interrupt handler on the lender-side. Whenever the device triggers an interrupt, the lender must notify the borrower, which in turn notifies the hypervisor, using eventfd. This method is not ideal, as the latency of triggering an interrupt is increased. A benefit, however, is that it allows us to enable legacy interrupts for devices borrowed by a VM, which is currently not supported when the borrower is a physical machine.

6 MULTI-DEVICE INTEROPERABILITY

Some processing workloads may require the use of multiple IO devices, and moving data between them in an efficient manner. This often involves the use device-to-device DMA, as described in Section 2.1, where a device is able to read from or write to the memory regions of other devices. However, as IOMMUs introduce a virtual address space for devices, TLPs must be routed through the root of the PCIe tree in order for the IOMMU to resolve virtual addresses. This means that peer-to-peer transactions directly between devices in the fabric is not possible when using an IOMMU. PCI-SIG has developed an extension to the transaction layer protocol that allows devices that have an understanding of IO virtual addresses to cache resolved addresses [20], but this is not widely available as it requires hardware support in devices.

Because of this, the general perception among device vendors and driver developers has become that in order to make peer-to-peer transactions work, the IOMMU must be disabled. This has led to a situation where device drivers would indiscriminately use physical addresses when setting up peer-to-peer access between devices. For our original Device Lending implementation, this posed a challenge, as we rely on intercepting calls made by the device driver to inject our own mappings in order to make DMA across the NTB transparent. However, this changed with the 4.9 version of the Linux kernel, when the DMA API was extended with a unified method for setting up mappings between devices. This extension makes it possible for Device Lending to intercept when a device is mapping another device's memory regions.

However, as devices installed in different hosts reside in different address space domains, the local IO address used by one host to reach a remote device is not the same address a different host would use to reach the same device. In order for a borrowed device, *source*, to reach another borrowed device, *target*, the borrower needs a mechanism to resolve virtual IO addresses it uses to addresses that *source*'s lender would use to reach *target*. As such, our solution is as follows:

- If *target* is local to the borrower, setting up a mapping is trivial. The lender simply sets up DMA windows to the individual memory regions of *target*, similar to how it already has set up a DMA window to the borrower's RAM. The lender returns the local IO addresses it would use to reach over the NTB to the memory regions of *target*. Note that this would work for any device in the borrower, not only those that are controlled by Device Lending.
- If *target* is locally installed in the same host as *source* (same lender), the lender simply sets up a local IOMMU mapping and returns the local IO addresses to the memory regions of *target*.
- If *target* is a remote device (different lenders), the *source*'s lender creates DMA windows through the appropriate NTB to *target*'s lender. Note that this NTB may be different to the one used in order to reach the borrower. It then returns the memory addresses it would use to reach over the NTB to the memory regions of *target*.

The borrower, after receiving these lender-local IO addresses, stores them along with its own virtual addresses to the memory regions of *target*. When the device driver using *source* calls the new DMA API functions to map the memory regions of *target* for *source*, we are able to look up the corresponding lender-local addresses and inject these. The driver can in turn initiate DMA, completely unaware of the location of both *source* and *target*, and the transfer will reach *target* through the correct NTB.

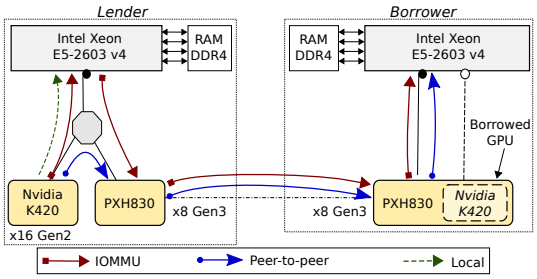


Figure 5: Configuration used in our IOMMU evaluation. The borrower is using the remote GPU. When the lender-side IOMMU is enabled, TLPs are routed through the lender's root before going over the NTB. We have also compared with a local instance, running on the lender itself.

7 PERFORMANCE EVALUATION

In this section, we evaluate the performance of our extensions to Device Lending. As our newly added virtualization support require the use of a lender-side IOMMU, we focus on the impact that IOMMU address virtualization has on performance. With support for multi-device interoperability, we have also evaluated the performance of peer-to-peer transfers. For our evaluations, we use bandwidth and latency as our performance metrics, as these two are the most commonly used for comparing interconnects.

7.1 IOMMU performance penalty

Since IOMMUs create a virtual address space, TLPs need to be routed through the root of the PCIe tree in order to resolve virtual IO addresses, effectively disabling peer-to-peer transfers. Processor designs are complex and often not well-documented, making it difficult to determine what exactly happens with the memory operations in progress once they leave the PCIe complex and enter the CPUs. Memory operations may be buffered, awaiting IOMMU translations, or the IOMMU may need to perform a multi-level table look up for resolving addresses.

TLPs are either *posted* or *non-posted* operations, meaning that some transactions, such as memory reads, require a completion. Read requests are affected by the number of hops in the path between requester and completer; the longer the path, the higher the request-completion latency becomes. As the number of read requests in flight is limited by how many uncompleted transactions a requester is able to keep open, a longer path can potentially reduce performance. In addition, PCIe allows a completer to respond with less data at the time than is actually requested. For example, a read TLP requesting 256 bytes may terminate with 4 completions containing 64 bytes each, rather than a single completion with 256 bytes.

In order to isolate the consequence of TLPs being routed through the root, we have used the setup shown in Figure 5. Two Intel Xeon machines are connected together with Dolphin's PXH830 NTB host

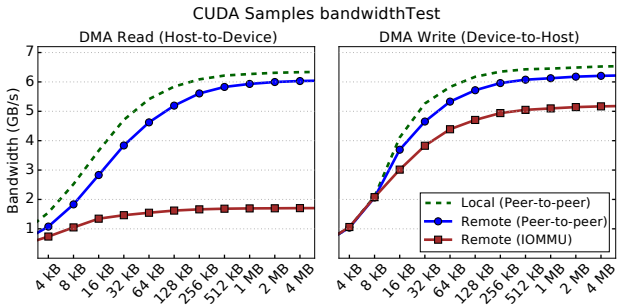


Figure 6: Reported bandwidth for different transfer sizes.

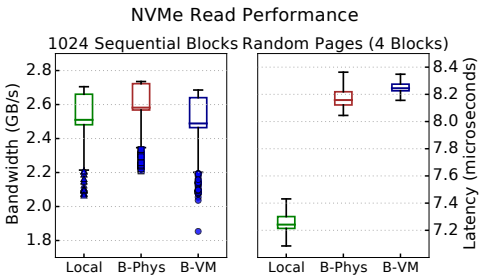


Figure 7: Bandwidth and latency when reading from disk (DMA write). We read 1024 sequential blocks for measuring bandwidth, and 4 blocks with a random offset for latency.

adapters [8] and an external x8 PCIe cable. The lender has a PCIe switch on the motherboard, with both the NTB adapter and an Nvidia Quadro K420 GPU sitting below it. Note that since the K420 is Gen2 x16, we only need a Gen3 x8 link between the NTB adapters, as they provide approximately the same bandwidth.

For this evaluation, we have chosen to create a high-bandwidth workload using the *bandwidthTest* [18] program. This utility program is from the CUDA Toolkit samples. Choosing this program serves an additional purpose, demonstrating that Device Lending truly works with remote devices, without requiring changes to application or driver software. The bandwidth is measured running on the borrower, using the remote K420's onboard DMA engine to copy data between GPU memory and borrower's RAM. For each transfer size, *bandwidthTest* initiates 100 transfers and then report the average bandwidth.

Figure 6 shows the reported average bandwidth for both DMA writes and DMA reads, comparing the performance of shortest path (peer-to-peer) with TLPs being routed through the root (IOMMU). We observe that the reported bandwidth is reduced when the IOMMU is enabled, especially for the read performance. As mentioned, a PCIe completer is allowed to reply with multiple completions to a single request. In our case, using a PCIe tracer similar in concept to that of network packet tracers, we observe that the read TLPs are actually modified by the lender-side *CPUs* (and not the completer). The maximum TLP payload size in our configuration is 256 bytes, meaning that devices can write or read up to 256 bytes per request. We observe, however, that every 256 byte request routed through the root is changed into 4×64 byte read requests before they are sent over the NTB. As read performance is already limited by the number of requests they are able to keep open, already changing the request size at the local side leads to less data being requested at the time, which again leads to very poor utilization of the link. Although not as bad as reads, write performance is also affected when the lender-side IOMMU is enabled.

Note that we have also compared our results to running locally on the lender, without using Device Lending. The achieved bandwidth of the local run is slightly better than our peer-to-peer performance, especially for the smaller transfer size; this is most likely due to the fact that the GPU sits physically farther away from the CPU running the driver, and therefore slightly increasing the time it takes to initiate a DMA transfer as well as other synchronization with the devices. We observe that for sizes of 1 megabyte and more, the significance of this additional latency decreases.

7.2 Pass-through comparison

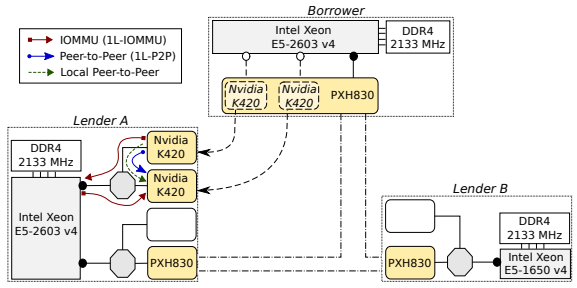
We have evaluated our KVM implementation using an Intel Optane 900P NVMe disk on a local machine without using Device Lending, a physical borrower (B-Phys), and from a VM guest (B-VM). The machines are connected back-to-back using PXH830 NTB adapters [8]. The RAM-to-RAM latency was measured to 550-580 nanoseconds, where the NTB adds around 350-370 nanoseconds. We have used QEMU 2.10.1 as our VM emulator, and running Ubuntu 17.04 LTS as the guest OS. Note that while any guest OS would be possible, including Microsoft Windows, we have chosen Linux in order to run the same benchmarking code on a physical borrower, as well as locally on the lender.

Figure 7 shows the bandwidth for reading 1024 sequential blocks repeated 1000 times. One block is 512 bytes. There is very little difference in the achieved bandwidth, except for a few additional outliers for our VM borrower (B-VM). Interestingly, we observe that the physical borrower (B-Phys) achieves slightly higher median bandwidth than the local comparison.

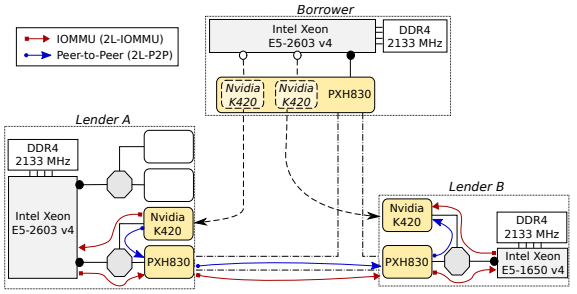
Latency was measured by reading 4 blocks repeated 10,000 times, each time at a random offset. Here, we observe that the difference between running locally and on the physical borrower is an increase in a little less than 1 microsecond. As the device now sits remotely, it has to first reach over the NTB once in order to retrieve the IO commands, and then reach over the NTB again in order to post the IO completion. This adds 700-730 nanoseconds to the latency, and is therefore an expected increase. We observe that passing the disk to a VM running on the borrower (B-VM), only increases the latency slightly compared to the physical borrower (B-Phys).

7.3 Device-to-device evaluation

In order to evaluate our multi-device support, we have evaluated the performance of device-to-device DMA transfers between two Nividia Quadro K420 GPUs. Using the CUDA API [18], there are two ways of initiating DMA transfers. The first one is similar to the *bandwidthTest* program, using the `cudaMemcpy()` function with device-to-device semantics. Using this method, the *driver* initiates the DMA transfer. The other method is code running on one GPU that writes to another GPU's memory directly. We have therefore developed two CUDA programs, one using the first method to measure DMA bandwidth (similarly to *bandwidthTest*) and the other to measure latency between the GPUs using the second method. Through CUDA's unified memory model, it is possible for the GPUs



(a) Two GPUs borrowed from the same lender.



(b) Two GPUs borrowed from different lenders.

Figure 8: The 3-node cluster configurations used in our multi-device evaluation, showing the data path for direct device-to-device transactions.

to access memory residing in RAM, without needing to explicitly copy it to GPU memory. Our two programs therefore also support this option, where one GPU first must write to the borrower's RAM, and then the other GPU must read from the borrower's RAM. Note that we do not use any special semantics in order to make our CUDA programs work for remote borrowed GPUs, they simply appear to the CUDA driver as if they are locally installed.

Figure 8 shows the two different configurations used in this evaluation, with the direct device-to-device data paths highlighted. Two GPUs are installed either in the same lender (Figure 8a), or in different lenders (Figure 8b). The machines are connected together using the PXH830 NTB adapter in a three-way configuration, providing a separate Gen3 x8 link between all three machines. The K420 GPUs are Gen2 x16, which is roughly the same bandwidth as Gen3 x8. Note that we have also included a peer-to-peer comparison, by running our same programs on Lender A.

As part of our evaluation, we have also evaluated the performance when memory buffers accessed by the GPUs reside in the borrower's RAM. In these scenarios, one GPU has to first write (over the NTB) to the borrower's RAM, and then the other GPU must read from the borrower's RAM (also over the NTB). The different data paths are illustrated in Figure 9. Note that each additional "hop" in the total path adds additional latency to the overall completion time. To summarize, we have evaluated the bandwidth and latency performance for the scenarios listed in Table 1.

7.3.1 Bandwidth. Using `cudaMemcpy()` for initiating transfers and `cudaEventRecord()` for recording time before and after transfers, our bandwidth program measures the DMA bandwidth for

Name	Scenario	Mem.	IOMMU
Local	Two local GPUs installed in same machine as driver.	GPU	Disabled
1L-P2P	Two remote GPUs borrowed from the same lender.	GPU	Disabled
1L-IOMMU	Two remote GPUs borrowed from the same lender.	GPU	Enabled
2L-P2P	Two remote GPUs borrowed from different lenders.	GPU	Disabled
2L-IOMMU	Two remote GPUs borrowed from different lenders.	GPU	Enabled
1L-RAM-P2P	Two remote GPUs borrowed from the same lender.	RAM	—
2L-RAM-P2P	Two remote GPUs borrowed from different lenders.	RAM	Disabled
2L-RAM-IOMMU	Two remote GPUs borrowed from different lenders.	RAM	Enabled

Table 1: Scenarios used in our device-to-device evaluation.

different transfer sizes, as depicted in Figure 10. Each transfer size is repeated 10,000 times, and we have plotted the median. The filled-out areas show the 1st to 99th percentiles, demonstrating that the variance between multiple runs is very low.

Comparing 1L-P2P and the local comparison in the top plot, the DMA bandwidth for smaller transfer sizes are affected by the longer distance between driver and GPU. As transfer sizes become larger, this factor decreases in significance, and for transfers of 4 megabyte and above, it is negligible. As with *bandwidthTest* (Figure 6), which *also* uses CUDA events to record time, we suspect that the protocol used by the driver in order to synchronize the GPU involves the driver going back and forth over the NTB multiple times.

As seen in Figure 10, direct device-to-device transfer is a DMA write operation only. Therefore, the difference between peer-to-peer transfers and when the IOMMU is enabled is not so extreme as it would be for reads. 2L-IOMMU is affected by needing to traverse both Lender A's and Lender B's roots, achieving slightly lower bandwidth than 1L-IOMMU. We see that when peer-to-peer transfers are possible (2L-P2P), the bandwidth is not significantly affected by having to traverse the NTB.

For transfers accessing the borrower's memory, however, the situation is quite different, as illustrated in Figure 10. As one GPU has to first write to borrower's RAM, before the other GPU can read from RAM, the read operation is the most significant performance factor. The performance is comparable to DMA reads shown in Figure 6, where routing read TLPs through the root appears to drastically reduce the link utilization because the read requests are altered. Peer-to-peer transactions that do not cross the root achieve a little under 6 GB/s (2L-RAM-P2P), which is the maximum expected for reads. Note that in the 1L-RAM-P2P scenario, traffic would traverse the same path regardless of the IOMMU being enabled or not (as depicted in Figure 9). We observe that this achieves the exact same performance as 2L-RAM-IOMMU, indicating that routing reads through the root generally leads to poor performance, and is not (exclusively) related to the use of IOMMUs.

7.3.2 Latency. We have also measured the ping-pong latency between two GPUs through CUDA's peer model. One GPU is tasked with increasing a counter, writing it to the other GPU's memory and waiting for an acknowledgement before continuing. The other GPU waits for the counter to increase by one, and acknowledges the increase by writing back the first GPU's memory. This process of counting upwards is repeated 100,000 times. For every step,

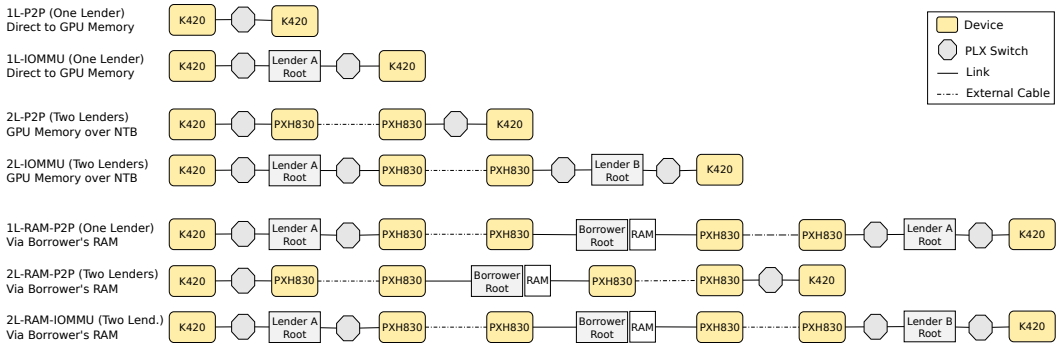


Figure 9: Data paths for the different scenarios. Each hop slightly increases the completion latency.

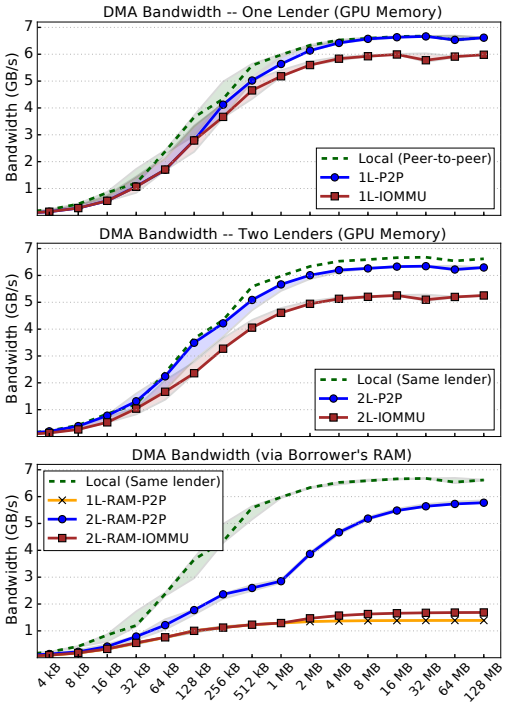


Figure 10: Median DMA bandwidth for different transfer sizes. The filled-out area represents the distribution between the 1st and 99th percentile for 10,000 runs. The local comparison is included in all three plots.

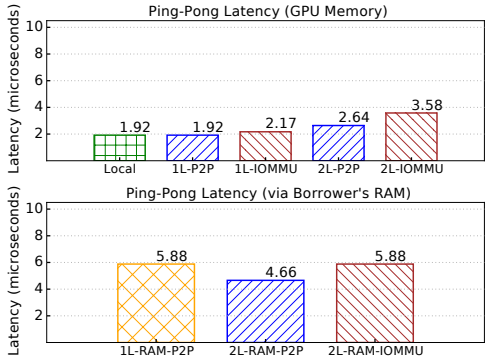


Figure 11: 99th percentiles of ping-pong latencies.

the current GPU clock cycle count is recorded and divided by the GPU's clock frequency. This provides us with an alternative to `cudaEventRecord()` for recording elapsed time, and we avoid any delay caused by explicit synchronization. We measured the RAM-to-RAM memory latency between the borrower and lender B to around 700 nanoseconds, where the NTB adds 350-365 nanoseconds.

Figure 11 shows the 99th percentile of ping-pong latencies for 100,000 repeated runs. The distribution between different runs is very low (less than 25 nanoseconds between minimum and maximum observed latency for each scenario). Using our alternative time recording eliminates additional access latency in the synchronization protocol between driver and GPU. When GPUs reside behind the same switch (1L-P2P), we achieve the same latency as for our local comparison. As the data paths increase, the latencies increase as well. We see that the latency for 2L-P2P increases with a little more than 700 nanoseconds, compared to 1L-P2P. This corresponds with the 350 nanoseconds added by the NTB (in one direction). For the scenarios where the memory buffers are hosted in the borrower's RAM, the latency increases significantly. Since their paths are the same, 1L-RAM-P2P and 2L-RAM-IOMMU have the same latency.

8 DISCUSSION AND CONCLUSION

In this paper, we presented our implementation for supporting interoperability between remote devices. As part of our work, we evaluated the impact of IO address virtualization on performance. Specifically, we have shown how lender-side IOMMUs affect the data path in terms of latency and bandwidth. As observed in our evaluations, longer paths introduce some additional latency for TLPs. When the driver and the device frequently communicate with each other, as seen in our GPU bandwidth evaluations, it may affect performance as TLPs has to go back and forth over the NTB. For device-to-device transfers that do not require driver synchronization, as is the case for our ping-pong latency evaluation, the distance between GPUs and driver is insignificant. It should be noted that traversing the NTB adds less than half of the latency added by InfiniBand FDR adapters [16, 25]. We have shown that Device Lending works without adding any performance overhead beyond what is expected of longer PCIe paths and the interconnect.

A major performance bottleneck occurs when DMA read requests are routed through the root, as the Intel Xeon CPUs used in our evaluation alter the requests in a way that leads to decreased utilization of the PCIe links. We observed that this drastically reduces performance for some scenarios. However, this effect was

also observed when the IOMMU was not enabled as well, appearing to be a problem with routing through the root in general, and not specifically related to IOMMU address translation. As our KVM implementation relies on the lender-side IOMMU, it is worth investigating further by evaluating other CPU architectures that implement an IOMMU, such as AMD EPYC/Zen and IBM POWER. Additional benefits to using the IOMMU include lenders isolating devices in their own domains, and remapping NTB mappings to lower memory for devices that do not support the entire 64-bit address space. For non-VM borrowers, routing through the root can be avoided by using PCIe switches and peer-to-peer transactions.

Additionally, our evaluation also demonstrates that it is possible to use remote IO resources without requiring *any* special semantics in application code or support in device drivers. We argue that being able to run the exact same code using remote GPUs as if they were locally installed, thus making use of one of the most complex GPU drivers on the market, demonstrate the strength of Device Lending compared to other approaches to distributed IO.

Finally, we have also presented how we have extended Device Lending with support for passing through borrowed remote devices for the KVM hypervisor. We have passed through a remote SSD to a VM guest, achieving the same bandwidth as the disk was locally installed and only slightly higher latency than that of a disk borrowed by a physical machine. Having built the infrastructure for this, we are currently investigating if a malicious VM can break out of the VM isolation by misusing Device Lending. Another candidate for further investigation is if possible to migrate VM instances running on one host to another with borrowed devices being passed-through. With our VM support and multi-device support, it is possible to offer highly customizable configurations of passed through remote devices, and dynamically reassign devices in order to optimize resource utilization.

ACKNOWLEDGMENTS

This work has been performed mainly in the context of the BIA project *PCIe* (#235530) funded by the Research Council of Norway (RCN), with contributions from the *LADIO* project (EU H2020 #731970). The authors would like to thank Kristoffer Robin Stokke for feedback on the manuscript. We also thank Stig Baugstø, Roy Nordstrøm and Hugo Kohmann at Dolphin Interconnect Solutions AS.

REFERENCES

[1] [n. d.]. Linux IOMMU Support. Retrieved April 28, 2018 from https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt

[2] [n. d.]. VFIO - "Virtual Function I/O". Retrieved April 28, 2018 from https://www.kernel.org/doc/Documentation/vfio.txt

[3] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajes Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Weigert. 2006. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* 10, 03 (2006).

[4] Knut Alnæs, Ernst H. Kristiansen, David B. Gustavson, and David V. James. 1990. Scalable Coherent Interface. In *Proceedings of International Conference on Computer Systems and Software Engineering (CompEuro)*. 446–453.

[5] Chelsio Communications Inc. 2015. The Case Against iWARP. Retrieved April 28, 2018 from https://www.chelsio.com/wp-content/uploads/resources/iWARP-Myths.pdf

[6] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A network stack for rack-scale computers. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 551–564.

[7] Alexandros Daglis, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2015. Manycore network interfaces for in-memory rack-scale computing. *ACM SIGARCH Computer Architecture News* 43, 3 (2015), 567–579.

[8] Dolphin Interconnect Solutions AS. [n. d.]. PXH830 Gen3 PCI Express NTB Host Adapter. Retrieved March 1, 2018 from http://www.dolphinics.no/products/PXH830.html

[9] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of International Conference on High Performance Computing and Simulation (HPCS)*. 224–231.

[10] T. Fountain, A. McCarthy, and F. Peng. 2005. PCI Express: An Overview of PCI Express, Cabled PCI Express and PXI Express. In *Proceedings of International Conference on Accelerator & Large Expt. Physics Control Systems (ICALEPCS)*.

[11] John P Hayes, Trevor Mudge, Quentin F Stout, Stephen Colley, and John Palmer. 1986. A Microprocessor-based Hypercube Supercomputer. *IEEE Micro* 6, 5 (1986), 6–17.

[12] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md Wasi-Ur-Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. 2012. High-performance design of hbase with RDMA over InfiniBand. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*. 774–785.

[13] Neo Jia and Kirti Wankhede. [n. d.]. VFIO Mediated Devices. Retrieved April 29, 2018 from https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt

[14] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, D K Panda, W Gropp, and R Thakur. 2004. High performance MPI-2 one-sided communication over InfiniBand. In *Proceedings of International Symposium on Cluster Computing and the Grid (CCGrid)*. 531–538.

[15] Lars Bjørlykke Kristiansen, Jonas Markussen, Håkon Kvale Stensland, Michael Riegler, Hugo Kohmann, Friedrich Seifert, Roy Nordstrøm, Carsten Griwodz, and Pål Halvorsen. 2016. Device Lending in PCI Express Networks. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*. 10:1–10:6.

[16] Mellanox Technologies. 2017. RoCE vs. iWARP Competitive Analysis. Retrieved April 28, 2018 from http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf

[17] NVIDIA Corporation. [n. d.]. Nvidia Virtual GPU Technology (vGPU). Retrieved April 28, 2018 from http://www.nvidia.com/object/virtual-gpus.html

[18] NVIDIA Corporation. 2017. CUDA Toolkit Documentation 9.1.85. Retrieved April 29, 2018 from http://docs.nvidia.com/cuda/

[19] Peripheral Component Interconnect Special Interest Group (PCI-SIG). 2008. *Multi-root I/O Virtualization and Sharing Specification.* https://www.pcisig.com/specifications/iov/multi-root/

[20] Peripheral Component Interconnect Special Interest Group (PCI-SIG) 2009. *Address Translation Services Revision 1.1.* Peripheral Component Interconnect Special Interest Group (PCI-SIG). https://www.pcisig.com/specifications/iov/ats/

[21] Peripheral Component Interconnect Special Interest Group (PCI-SIG). 2010. *PCI Express 3.1 Base Specification.* https://pcisig.com/specifications

[22] Peripheral Component Interconnect Special Interest Group (PCI-SIG). 2010. *Single-root I/O Virtualization and Sharing Specification.* https://www.pcisig.com/specifications/iov/single-root/

[23] Murali Ravindran. 2008. Extending Cabled PCI Express to Connect Devices with Independent PCI Domains. In *Proceedings of the 2nd annual IEEE Systems Conference (SysCon)*. 1–7.

[24] Jack Regula. 2004. *Using Non-transparent Bridging in PCI Express Systems.* PLX Technology, Inc. White paper.

[25] Davide Rosetti. 2014. Benchmarking GPUDirect RDMA on Modern Server Platforms. Retrieved April 29, 2018 from http://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/

[26] Kazuo Saito, Koji Anai, Keiju Igarashi, Takeshi Nishikawa, Ryoichi Himeno, and Kazuhiro Yoguchi. 1998. ATM bus system. US patent No. 5,796,741 A.

[27] Mark J. Sullivan. 2010. *Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge.* Technical Report. Intel Corporation.

[28] Jun Suzuki, Yoichi Hidaka, Junichi Higuchi, Teruyuki Baba, Nobuharu Kami, and Takashi Yoshikawa. 2010. Multi-root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In *Proceedings of Symposium on High Performance Interconnects (HOTI)*. IEEE, 25–31.

[29] A Trivedi, B Metzler, and P Stuedi. 2011. A case for RDMA in clouds. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*. 17:1–17:5.

[30] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. 2013. Secure I/O Device Sharing Among Virtual Machines on Multiple Hosts. *ACM SIGARCH Computing Architecture News* 41, 3 (2013), 108–119.

[31] A. Venkatesh, H. Subramoni, K. Hamidouche, and Dhabaleswar K. Panda. 2014. A high performance broadcast design with hardware multicast and GPUDirect RDMA for streaming applications on Infiniband clusters. In *Proceedings of International Conference on High Performance Computing (HiPC)*.

[32] Colin Whitby-Strevens. 1985. The transputer. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 292–300.

[33] Heymian Wong. [n. d.]. *PCI Express Multi-Root Switch Reconfiguration During System Operation.* Master's thesis. Massachusetts Institute of Technology.