UiO **University of Oslo**

Jonas Sæther Markussen

SmartIO: Device sharing and memory disaggregation in PCIe clusters using non-transparent bridging

Dissertation submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

Dolphin Interconnect Solutions

Simula Research Laboratory

2022

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2561

ISSN 1501-7710

Print production: Graphics Center, University of Oslo.

Abstract

Distributed and parallel computing applications are becoming increasingly compute-heavy and data-driven, accelerating the need for disaggregation solutions that enable sharing of I/O resources between networked machines. For example, in a heterogeneous computing cluster, different machines may have different devices available to them, but distributing I/O resources in a way that maximizes both resource utilization and overall cluster performance is a challenge. To facilitate device sharing and memory disaggregation among machines connected using PCIe non-transparent bridges, we present SmartIO. SmartIO makes all machines in the cluster, including their internal devices and memory, part of a common PCIe domain. By leveraging the memory mapping capabilities of non-transparent bridges, remote resources may be used directly, as if these resources were local to the machines using them. Whether devices are local or remote is made transparent by SmartIO. NVMes, GPUs, FPGAs, NICs, and any other PCIe device can be dynamically shared with and distributed to remote machines, and it is even possible to disaggregate devices and memory, in order to share component parts with multiple machines at the same time. Software is entirely removed from the performance-critical path, allowing remote resources to be used with native PCIe performance. To demonstrate that SmartIO is an efficient solution, we have performed a comprehensive evaluation consisting of a wide range of performance experiments, including both synthetic benchmarks and realistic, large-scale workloads. Our experimental results show that remote resources can be used without any performance overhead compared to using local resources, in terms of throughput and latency. Thus, compared to existing disaggregation solutions, SmartIO provides more efficient, low-cost resource sharing, increasing the overall system performance and resource utilization.

Acknowledgements

My sincere gratitude goes to my friend and colleague, Lars Bjørlykke Kristiansen. This dissertation would simply not have been possible without your support and help. Thank you for sharing my frustration over confusing lab results, the many hours spent teaching me about technical details, as well as your countless insights, technical understanding, and visionary ideas for things to try out and experiments to do. I'm looking forward to working with you for many years to come.

I'd also like to thank to my doctoral advisors, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. Not only have you been extremely patient with me, more than anyone can reasonably expect from any advisor, but your invaluable feedback and input has truly elevated the quality of my work. Your innovative approach to computer science and dedication to high quality research inspires me, and I hope we may collaborate on future research projects. I also want to thank all of my colleagues at Dolphin, especially my bosses Hugo Kohmann and Roy Nordstrøm, who have allowed me time and resources to perform this work—even when it didn't align perfectly with tasks that were more urgent for the company.

Thanks to my family and many supportive friends, particularly Katrine, Anders, Christian, Martin, Axel, Morten, Thomas, Karoline, Nora, Jonatan, and Kristian. Thank you for continuing to believe in me, even when I doubted myself. It is because of your continued support I didn't give up a long time ago.

I'd also like to thank my friends and former colleagues at Simula: Michael, Bendik, Halvor, Konstantin, Kristoffer R., and Preben. You made the working environment of Simula educational and also very fun for me. I have many fun memories from conference trips abroad, general office humor, and playing pool and joking around. Finally, I would like to thank the Simula Premium Coffee Club (aka. "The Secret Society of Aromatic Snobbery") for providing the excellent coffee consumed at many late hours as deadlines were approaching, thus saving me from the awful, black substance produced by the (so-called) "coffee" machine.

Jonas Sæther Markussen
Oslo, May 2022

List of papers

Paper I

Lars Bjørlykke Kristiansen, Jonas Markussen, Håkon Kvale Stensland, Michael Riegler, Hugo Kohmann, Friedrich Seifert, Roy Nordstrøm, Carsten Griwodz, and Pål Halvorsen. “Device Lending in PCI Express Networks.” In: *Proceedings of the 26th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV’16. May 2016, 10:1–10:6. DOI: 10.1145/2910642.2910650

Paper II

Konstantin Pogorelov, Michael Riegler, Jonas Markussen, Håkon Kvale Stensland, Pål Halvorsen, Carsten Griwodz, Sigrun Losada Eskeland, and Thomas de Lange. “Efficient Processing of Videos in a Multi Auditory Environment Using Device Lending of GPUs.” In: *Proceedings of the 7th ACM International Conference on Multimedia Systems*. MMSys’16. May 2016, pp. 381–386. DOI: 10.1145/2910017.2910636

Paper III

Jonas Markussen, Lars Bjørlykke Kristiansen, Håkon Kvale Stensland, Friedrich Seifert, Carsten Griwodz, and Pål Halvorsen. “Flexible Device Sharing in PCIe Clusters Using Device Lending.” In: *Proceedings of the 47th ACM International Conference on Parallel Processing Companion*. ICPP’18 Comp. August 2018, 48:1–48:10. DOI: 10.1145/3229710.3229759

Paper IV

Jonas Markussen, Lars Bjørlykke Kristiansen, Rune Johan Borgli, Håkon Kvale Stensland, Friedrich Seifert, Michael Riegler, Carsten Griwodz, and Pål Halvorsen. “Flexible Device Compositions and Dynamic Resource Sharing in PCIe Interconnected Clusters using Device Lending.” In: *Cluster Computing* vol. 23, no. 2 (June 2020), pp. 1211–1234. ISSN: 1573-7543. DOI: 10.1007/s10586-019-02988-0

Paper V

Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. “SmartIO: Zero-overhead Device Sharing through PCIe Networking.” In: *ACM Transactions on Computer Systems* vol. 23, no. 1–2 (July 2021), 2:1–2:78. ISSN: 1573-7543. DOI: 10.1145/3462545

Contents

List of figures

List of abbreviations

API application programming interface 9, 10, 15, 30, 35, 51, 55

ATS Address Translation Services 60

BAR Base Address Register 19–22, 25–28, 30, 32, 34, 35, 51, 52, 58,
Glossary: Base Address Register (BAR)

BIOS Basic Input/Output System 8

CPU central processing unit 3–5, 8, 14, 15, 19–22, 26, 32, 36, 39, 40, 42, 45, 48–50, 57, 58, 60

CQ completion queue 32, 36, 44, 46

CXL Compute Express Link 61

DMA direct memory access 4, 5, 8, 9, 16, 19–22, 25–30, 32, 37, 39, 41–44, 49, 53, 56, 57, 60,
Glossary: direct memory access (DMA)

FIO Flexible I/O tester 39

FPGA field-programmable gate array 1, 14, 53

GPU graphics processing unit 1–3, 9, 11, 13–16, 21, 23, 31, 33–38, 41–47, 49, 50, 53, 59

I/O input/output 1–3, 6–9, 12–16, 22, 23, 25, 26, 28–30, 32, 36, 38, 41, 44–50, 53, 54, 57, 59, 60,
Glossary: input/output (I/O)

IOMMU I/O Memory Management Unit 2, 16, 22, 27–30, 37, 39, 43, 46, 54, 56, 57, 60,
Glossary: I/O Memory Management Unit (IOMMU)

KVM Linux kernel-based virtual machine hypervisor 11, 14, 28, 30, 54, 55,
Glossary: hypervisor

MDEV Mediated Device Driver 14–17, 24, 25, 28–31, 37, 38, 44, 49–52, 54–58, 60,
Glossary: Mediated Device Driver (MDEV)

MMIO memory-mapped I/O 20, 25, 30,
Glossary: memory-mapped I/O (MMIO)

MR-IOV Multi-Root I/O Virtualization 48, 50, 58

MSI message-signaled interrupts 20, 21, 27,
Glossary: message-signaled interrupts (MSI)

NIC network interface card 2, 9, 11, 14, 23, 37, 49, 53, 59

NTB non-transparent bridge 4–7, 9–12, 20–30, 38–40, 43, 46, 51–56, 58, 59,
Glossary: non-transparent bridge (NTB)

NVMe non-volatile memory express storage device 3, 9, 11, 13–16, 31–41, 44–49, 53, 54, 58, 59, 61

NVMe-oF NVMe over Fabrics 3, 38, 49

OS operating system 2, 3, 7, 8, 10–12, 26–28, 55, 56

PASID Process Address Space ID 60

PCIe Peripheral Component Interconnect Express 2–11, 13–17, 19–21, 23–25, 27, 28, 31, 33, 35, 37–41, 43, 47–61

RAM random access memory 19–22, 26, 29, 30, 34, 36, 37, 41, 43, 44, 47, 51, 56–58

RDMA remote direct memory access 2, 3, 6, 8, 15, 16, 37, 38, 48–50, 56,
Glossary: remote direct memory access (RDMA)

SISCI Software Infrastructure for Shared-Memory Cluster Interconnects 10, 15, 24, 30–32, 47, 49–51, 55, 56

SQ submission queue 32, 36, 44–46

SR-IOV Single-Root I/O Virtualization 3, 9, 15, 23, 31, 37, 48–51, 58,
Glossary: Single-Root I/O Virtualization (SR-IOV)

SSD solid-state flash memory storage device 41

VF virtual device function 9, 23, 31, 48, 49, 51,
Glossary: Single-Root I/O Virtualization (SR-IOV)

VFIO Virtual Function I/O
see also pass-through

VM virtual machine 2, 6–8, 10, 11, 13, 14, 17, 22, 23, 28–30, 37, 50–52, 54–57, 60

Chapter 1

Introduction

Cluster computing applications often have high requirements to I/O performance. For example, many computing clusters rely on compute accelerators, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), to increase the processing speed. In recent years, we have also seen a convergence of the high-performance computing, big data, and machine learning research fields. This development has led to new demands to I/O performance where fast access to high-volume storage devices is becoming a requirement for high-performance computing, while low latency networking and making use of compute accelerators have become cloud computing issues [8, 52, 55]. If I/O resources (devices) are scarcely distributed in the cluster, cluster machines with I/O resources may become bottlenecks, for example when a workload requires heavy computation on GPUs or fast access to storage. Contrarily, over-provisioning machines with resources may lead to devices becoming underutilized if a workload's I/O demands are more sporadic. Distributed processing workloads may even require a heterogeneous cluster design, with widely different compositions of devices and memory resources for individual machines in the cluster. Being able to share and partition devices between cluster machines at run-time leads to more efficient utilization, as individual machines may dynamically scale up or down I/O resources based on current workload requirements (Figure 1.1).

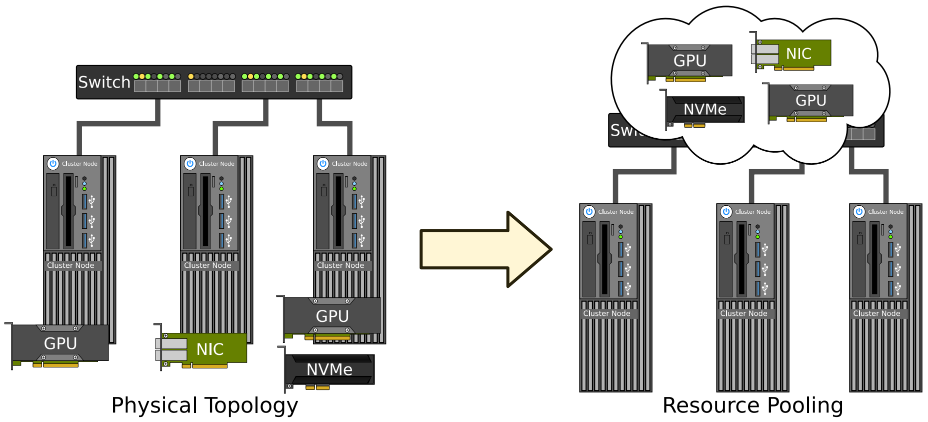


Figure 1.1: If machines could pool their internal devices, it would be possible to avoid queuing work on dedicated machines with particular device configurations. Instead, each machine could dynamically compose the I/O infrastructure needed to meet a workload's I/O requirements, by borrowing devices from other machines and releasing them when they are no longer needed.

In order to meet the latency and throughput requirements of data-driven and compute-heavy workloads, there is a need for flexible, yet efficient, sharing of I/O resources in computing clusters. This dissertation contributes to this goal by presenting a solution that enables distributing devices and sharing memory resources between machines interconnected with Peripheral Component Interconnect Express (PCIe) [42]. By leveraging memory mapping functionality supported by the PCIe networking hardware, we make it possible to use resources residing in remote machines as if they were installed in the same machine. Whether resources are local or remote is made transparent to application software, operating system (OS), and even device drivers, and remote resources can be used in a manner that is indistinguishable from using resources attached to the local PCIe bus. Existing device drivers and application software may use remote resources without requiring any adaptations. Not only does this make it easier to increase the overall resource utilization in the cluster, but it also becomes easier to design and implement distributed applications as software no longer needs to be written with accessing remote resources in mind, but can instead be implemented as if all resources are local. Using our solution, I/O resources are no longer locked to individual machines, and can instead be shared freely with other machines in the cluster.

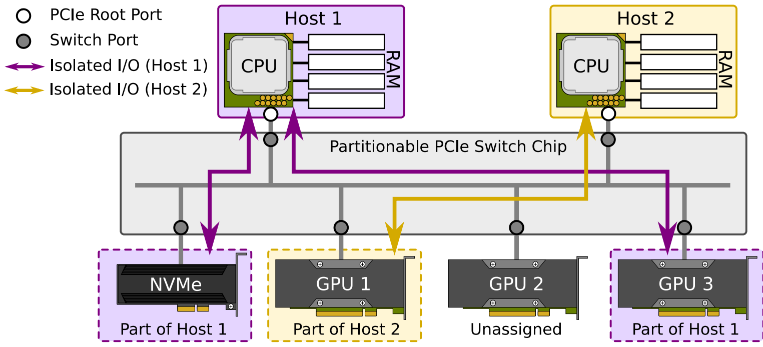
1.1 Background and motivation

In cloud computing environments, dynamically scaling resources is often accomplished through virtualization. Virtual machine (VM) hypervisors may dynamically add virtual I/O devices to VM instances on demand. It is even possible to temporarily suspend computation to migrate VMs to host machines with more hardware resources available, should the requirements of a VM guest exceed the available local resources. However, when the raw, bare-metal I/O performance is required, for example in the case of GPU-intensive machine learning workloads, resource virtualization may not be a viable solution. In this regard, it is possible to "pass through" physical I/O devices to a VM guest using an I/O Memory Management Unit (IOMMU). The IOMMU facilitates direct access to hardware from the guest without compromising the virtualized environment [1, 36]. As such, pass-through allows physical hardware to be used by a VM guest with minimal software overhead. However, as physical devices are tightly coupled with the host they are installed in, this pass-through technique suffers from a lack of flexibility. Distributing VMs across hosts in the network in a way that maximizes resource utilization and adapts dynamically to varying I/O requirements, without sacrificing the bare-metal performance that pass-through provides, remains a challenge.

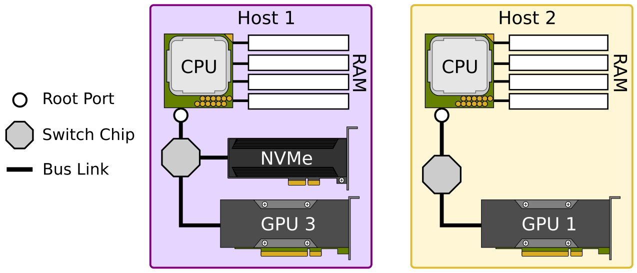
Another challenge is the networking technology itself. Despite having been a research topic for decades, moving data to remote computing units over a network remains a costly operation that introduces large performance overheads compared to using local resources. As such, many network interface cards (NICs) support zero-copy of application memory from one system to another through remote direct memory access (RDMA) [21]. RDMA is not only used in many distributed shared-memory cluster applications, but is also frequently used for implementing I/O resource disaggregation in software. For example, non-volatile memory express storage devices (NVMes) may be disaggregated and shared with remote systems with very low latency. This is the case for NVMe over Fabrics (NVMe-oF), where RDMA is used to provide direct access and avoid going through the block-layer of the OS on the server [19]. Similarly, the result of a GPU computation may be copied out of GPU memory and onto the network directly using RDMA, without being copied to system memory first and going through the network stack [63]. However, while RDMA allows data to be transferred efficiently over the network, translation between the network protocol and the local I/O bus is unavoidable. Compared to accessing a local device, this protocol translation incurs latency overheads that are not insignificant. Moreover, as RDMA requires the use of specific programming models like message-passing [23], disaggregation solutions based on RDMA are usually implemented either as application-specific middleware, or as part of the application itself. The sharing capabilities of RDMA solutions are, therefore, often limited to a single type of device. Sharing several types of devices, for example both GPUs and NVMes, usually requires multiple disaggregation implementations, and integrating them with each other may be a challenge.

Extending the PCIe bus out of a single computer system and using it as a high-speed interconnection technology is a compelling alternative to distributed I/O over a traditional network [16, 44, 45]. As PCIe is the standard for connecting I/O devices to a local computer system, using only native PCIe would have clear performance advantage, since conversion between network protocol and I/O bus would not be necessary. However, since PCIe was originally designed to connect devices to the local central processing unit (CPU) on a motherboard, individual computer systems operate with different PCIe address domains. Because of this, some PCIe switch chip hardware have virtualization support for dynamic partitioning [7, 35, 64]. Multiple CPUs can be connected to the same PCIe fabric by mapping partitions to the individual address domains of each CPU. Additionally, devices can be attached directly to the partitionable switch chip, rather than being owned by individual machines. This allows switch-attached devices to be logically assigned to different machines, as illustrated in Figure 1.2.

Nonetheless, because partitioning isolates CPUs in separate address domains, this approach does not make it possible for machines to share their *internal* resources. Memory, or other devices that are attached to the local PCIe bus and not the partitionable switch (chip), cannot be shared. Thus, partitioning lacks the shared-memory capabilities needed to support host-to-host communication over native PCIe, and other networking technologies, such as Ethernet or InfiniBand, must be used instead. Consequently, disaggregating devices and sharing them with multiple machines at the same time require either alternative methods, like RDMA, or additional virtualization support in the device itself, i.e., Single-Root I/O Virtualization (SR-IOV). While approaches using PCIe fabric partitioning can be said to enable a composable I/O infrastructure [7], they stop short of providing *networking* capabilities over PCIe.



(a) Partitioning allows CPUs and devices with different address domains to be isolated.



(b) Machines have separate (logical) PCIe device trees.

Figure 1.2: PCIe switch chips with partitioning support can be used to connect multiple CPUs and freestanding devices to a common PCIe fabric. However, as systems are isolated, shared memory communication over PCIe is not possible.

Due to its intrinsic memory addressing abilities and low latency overhead, it is desirable to use PCIe as the enabling networking technology for distributed, shared-memory communication [34, 48]. This requires translating memory transactions from one machine's address domain to another. By far, the most common way of translating addresses between different PCIe address domains is by using a special type of device called a non-transparent bridge (NTB) [5, 45, 61]. By using NTBs to implement plug-in host adapter cards and cluster switches, it becomes possible to connect independent computer systems as depicted in Figure 1.3. The memory address translation capabilities of NTBs make it possible for a machine to map (parts of) the address space of remote systems. Since all memory address translations are done in the NTB hardware, memory-to-memory transfers are supported with very low latency.

More interestingly, however, is the fact that in such NTB-based networks, all CPUs and internal PCIe devices are attached to the same, shared PCIe fabric. Remote resources, such as the internal memory and devices of other machines, could be mapped into a local system and accessed through the NTB with very little performance overhead. Similarly, a device capable of direct

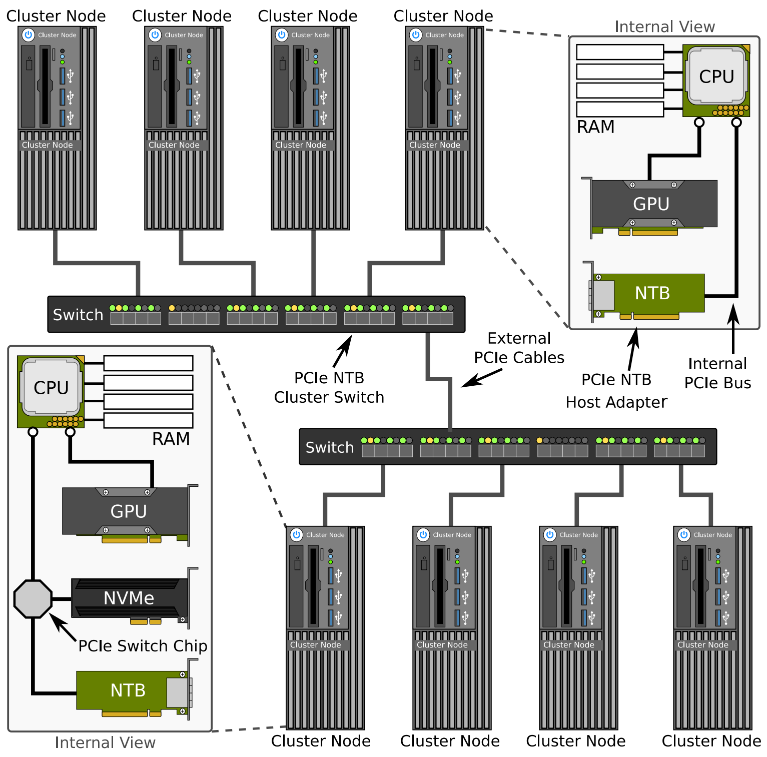
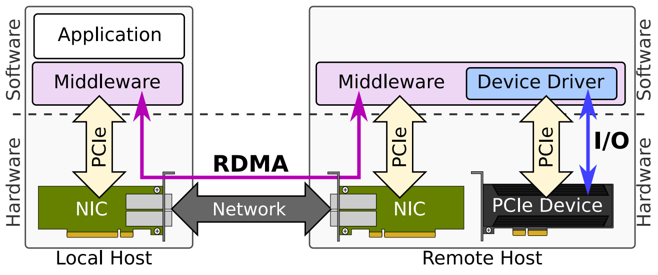


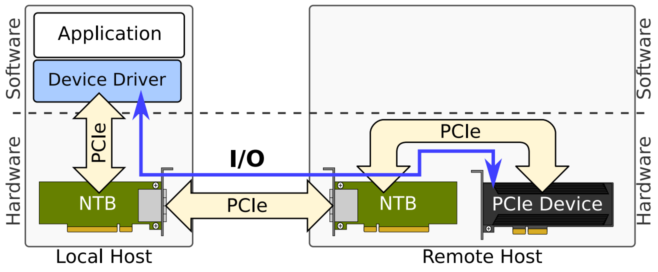
Figure 1.3: Example of a heterogeneous PCIe cluster with external PCIe links using adapter cards capable of NTB. The CPUs as well as internal devices of all cluster machines (nodes) are all attached to the same PCIe network fabric.

memory access (DMA) could also use the NTB to access remote resources. This approach eliminates the need to use memory on the remote system as an intermediate step when transferring data, and any software in the data transfer path can be avoided entirely, as shown in Figure 1.4. Rather than relying on dedicated servers or requiring that devices are attached to special switches, we could create distributed, peer-to-peer sharing system using NTBs; *all* machines in the cluster can share *all* of their resources, including internal PCIe devices and system memory. Moreover, as centralized servers can be avoided, the risks of individual machines becoming performance bottlenecks or single points of failure are reduced.

However, using an NTB to map remote resources requires awareness of the address space on the remote system. For example, a device driver must use addresses that correspond to the remote device's address space when initiating DMA transfers. Extensive modifications to device driver software would be required in order to manage multiple address space layouts. This is infeasible due to the vast amount of devices and device driver implementations that exist. Although we could use virtualization to hide the fact that devices are on the



(a) Accessing remote resources over traditional network using RDMA.



(b) Accessing remote resources over native PCIe using NTB.

Figure 1.4: Many distributed I/O solutions have performance overheads because they rely on middleware or other forms of software facilitation on the remote system. By setting up memory mappings through the NTB, remote hardware resources can be accessed directly without any software in the critical path.

other side of an NTB [60], by relying on VMs we would forgo the possibility of using bare-metal machines for processing. Hence, a realistic solution based on NTBs requires a mechanism for abstracting away the physical location of a resource, as well as the address space of the machine it is installed in, without requiring virtualization (although sharing to VMs should *also* be supported).

Nevertheless, this kind of abstraction gives rise to yet another challenge. A device driver that is unaware that a device is remote may assume that the entire local address space can be reached by the device. NTB maps must be in place *before* the driver interacts with the device, but predicting in advance which memory addresses a device driver may use is generally not possible. Deferring the action of mapping through the NTB until a time when addresses may be known is not realistic, as this would require synchronizing with the remote system and introduce communication overhead in the performance-critical path. A way to prepare the necessary memory-maps through the NTB, without adding communication latency, is needed.

1.2 Problem statement

Utilizing PCIe NTBs to share resources among machines in a PCIe-networked cluster requires a solution for abstracting away the physical location of a resource, including the address space of the computer system it is installed in. More specifically, as it is desirable to avoid modifications to existing device drivers and application software, such a solution must also be able to present resources to the system as if they were locally installed. Additionally, the solution must also allow remote resources to be used with the same performance expected for native PCIe, i.e., with the same performance as if they were attached to the local PCIe bus. Hence, the goal of this dissertation is to develop a framework for sharing and distributing I/O resources (devices and memory) in a way that makes it indistinguishable if a resource is remote or local. The challenges of this goal are addressed under the following research question:

> *Can NTBs be leveraged to allow the internal memory and devices of individual computers in a PCIe-networked cluster to be shared with and used by remote machines in the cluster, as if these resources were local to the remote machines?*

In particular, this research question can be broken down into the following six objectives:

Objective 1: Ubiquitous sharing in the cluster should be supported, allowing any machine to contribute any of its internal PCIe devices, and allowing any machine to be able to use shared devices, even contributing and using devices at the same time.

The main motivation for our goal of building a system for I/O resource sharing is to make it easier to scale out and use more resources than there are available in a single computer. If any standard PCIe device inside any machine could be shared with other machines, the I/O resource utilization in the cluster could be greatly increased. Additionally, by avoiding dedicated servers and allowing all computers in the cluster to participate in the sharing, contributing their own resources and using resources shared by others, we would effectively enable a distributed, peer-to-peer sharing model. This objective sets our goal apart from existing PCIe-based solutions, as these require a central server or devices that are directly attached to a PCIe switch.

Objective 2: The fact that resources may be remote should be functionally transparent, allowing systems to use remote resources in the same way as if they were local, without requiring any modifications to device hardware, device drivers, host OS, or application software.

If the solution could make remote devices behave as if they were locally installed, presenting resources to the system on a level "underneath" the OS, it would become possible to distribute devices to *physical* hosts as well, and not only VMs.

In other words, remote resources should appear as if they were part of the local PCIe device tree, and application software could make use of remote devices using native interfaces in the same way it would use local devices. Furthermore, by avoiding application- or device-specific middleware, and instead memory-mapping remote system and device memory directly, existing device drivers and even the host OS itself would be able to interact with remote resources natively. Avoiding any special adaptions to software would make scaling out significantly easier than what is currently possible with existing middleware-based solutions for distributed I/O, particularly those based on RDMA.

Objective 3: The fact that resources may be remote should be transparent with regard to performance, remote resources should be used with native PCIe performance, and as close to local access as possible.

Moving data to remote units over the network introduces large performance overhead compared to accessing local resources. In order to further blur the hard separation between "remote" and "local", remote resources should not only behave functionally as if they were locally installed in the system using them, but also have comparable performance. To achieve this, any communication overhead and intermediate data copying in the critical path must be completely avoided, a requirement that rules out (most) traditional methods of sharing resources over a network. Remote resources should be accessed directly over *native* PCIe, which would improve the overall I/O performance in the cluster.

Objective 4: Shared resources should be distributed *dynamically*, and direct access to device memory and system memory should be configured at run-time, also between multiple devices residing in different hosts.

As stated in Objective 2, the solution should work for physical hosts, and not only VMs. Therefore, it must be possible to assign and reassign resources while all machines in the cluster are running, without requiring rebooting hosts or changing settings in the BIOS. For devices, this introduces the requirement that the OS supports hot-adding devices to the system (which most modern OS implementations do). Not only would this would allow systems to dynamically scale up or down their I/O resources based on immediate workload requirements, but devices could be more efficiently partitioned between machines in the cluster, increasing the overall resource utilization. Furthermore, the solution should also be able to automatically discover resource location, without requiring that the user knows anything about the underlying PCIe network topology, and dynamically set up memory mappings between devices, CPUs, and memory resources. An example would be enabling PCIe peer-to-peer between two or more DMA-capable devices that are physically installed in different machines.

Objective 5: Disaggregation of system memory, device memory, and device *functionality* should be supported, and the solution should be able to distribute component parts to different hosts, as well as provide software facilities for resources that do not support disaggregation in hardware.

Because most device drivers are written in a way that assumes exclusive control over a device, some devices implement virtualization support in hardware, i.e., SR-IOV, that makes them appear to a system as having multiple virtual device functions (VFs). The solution should be able to disaggregate such SR-IOV-capable devices, and distribute their VFs to different machines, allowing multiple computers to use the same device simultaneously. However, since not all devices implement SR-IOV, the solution should also provide a device driver application programming interface (API) that will make it possible to disaggregate memory and device resources in software. In addition to the native sharing capabilities described in Objectives 1–4, this API would provide facilities for memory-mapping device registers as well as mapping shared memory segment for a DMA-capable device. Effectively, this would bring shared-memory concepts to device driver implementations, allowing device operation and device resources to become part of the same global address space as distributed cluster applications. This would allow multiple machines to simultaneously share the same, non-SR-IOV device, as well as making it possible to combine traditional I/O with PCIe cluster capabilities such as zero-copy data transfer and multicasting. Moreover, the API should be designed so that a driver implementation does not need to consider the system-local address space of the computer system where a device is installed, thus alleviating the complexity of programming device drivers for remote devices using NTBs.

Objective 6: To prove real-world deployment capabilities, the solution should be tested on realistic and relevant workloads and benchmarks.

In order to confirm that I/O resources can be distributed to, and shared with, remote machines, a comprehensive performance evaluation covering all components of the implementation is needed. As the solution should provide *native* PCIe performance (Objective 3), all parts should be thoroughly tested with latency and throughput in mind, in order to reveal any potential performance bottlenecks. Standardized test suites should be used as far as possible, to prove that application software really can be unmodified (Objective 2). Moreover, to demonstrate the completeness of the solution, the evaluation should also include workloads relying on different PCIe network topologies and include several types of devices, such as NVMes, GPUs, and NICs. Finally, a prototype device driver using the device driver API (Objective 5) should be developed and evaluated. This driver should demonstrate that it is possible to implement a distributed device driver, disaggregating a non-SR-IOV device in software, and sharing it with multiple machines. The driver should also demonstrate how it can rely on memory disaggregation and shared memory capabilities to implement data path optimizations.

1.3 Scope and limitations

The research presented in this dissertation contributed to a collaborative project between academia and industry, with Dolphin Interconnect Solutions (Dolphin) as the industry partner.[1] The goal of this project was to develop a new framework for sharing devices and memory resources among computer systems connected with PCIe. The scope of this dissertation is the implementation and performance evaluation of the fundamental mechanisms that make this framework possible: **the discovery, addressing, access, and use of remote PCIe-attached resources**. We aim to support this for three different levels of sharing:

1. Dynamically distributing devices to remote machines.
2. Dynamically distributing (physical) devices to VMs running on remote machines.
3. Enabling disaggregation of devices and memory resources in software, allowing them to be shared simultaneously by software processes running on several machines.

Exploring resource sharing possibilities using alternative networking technologies, such as InfiniBand, is outside the scope of our dissertation. The desired objectives for the sharing framework, as stated in Section 1.2, necessitate the use of PCIe NTBs. Dolphin's NTB adapter cards and cluster switches make it possible to build heterogeneous computing clusters in various network topologies. Figure 1.3 is an example of such a cluster. Memory mappings between machines are configured using Dolphin's existing NTB driver, and application software may use the Software Infrastructure for Shared-Memory Cluster Interconnects (SISCI) programming API to interact with this NTB driver in order to dynamically set up and tear down these memory mappings and implement shared-memory communication. In order to fit our work into this existing cluster networking foundation, we rely on Dolphin's NTB hardware and have implemented our sharing framework as part of their driver stack. This makes it possible to extend existing functionality, rather than developing an entirely new NTB driver from scratch. To enable software disaggregation, for example, we can add device-oriented semantics and device driver support functions to the already existing SISCI API. Even so, it should be pointed out that while our specific implementation is building on Dolphin NTBs, the ideas and concepts of our work are general, and can in fact be implemented for *any* NTB hardware that has similar capabilities.

Using our framework, both host machines and VMs should be able to use shared devices. In order to accomplish this, resources must to be presented to the system at the OS-level so that interactions with a device can be intercepted. This intercepting mechanism involves manipulation of device driver interfaces

[1] Funded by the Research Council of Norway under the "Unified PCI Express for Distributed Component Virtualization" project (RCN/NFR #235530), with additional contributions from the "LADIO: Live Action Data Input/Output" project (EU H2020 #731970).

and requires a detailed understanding of OS internals. For this reason, our framework is implemented for Linux, as its source code is available as open source and may be studied. For the same reason, we have also extended the Linux kernel-based virtual machine hypervisor (KVM hypervisor) in order to support sharing devices with VM guests. Regardless, we do not limit our work to a specific Linux distribution, as it is possible to support several different versions of the Linux kernel. It should also be noted that for VMs, there are no limitations for which OS may run in the guest.

As per our objectives, sharing of any PCIe device should be supported, from PCIe 1.0 devices up to PCIe 4.0 devices. Our evaluation includes a wide range of common (and commodity) devices, like NVMes, Ethernet NICs, and GPUs. However, the framework should also not require any modifications to existing device driver implementations or device hardware. While NTB clusters can be anywhere from 2 up to 60 machines, in many different network topologies, a limitation of the evaluation of our implementation is that it for the most part includes scenarios with cluster networks of only a few machines, since devices can only be used by one machine at the time. Nevertheless, more advanced network topologies and larger clusters (of up to 60 machines) are used in a few experiments, as our framework supports disaggregating devices in software, allowing devices to be used simultaneously by many machines.

Finally, it should be mentioned that several related research topics emerge from enabling the envisioned sharing framework this dissertation aims for. Examples include finding algorithms for scheduling workloads on different machines in the cluster in order to optimize resource utilization, or choosing the appropriate trade-off between fairness and workload performance requirements when provisioning resources to machines. The security implications of allowing remote machines to use and control internal system resources is also something that becomes relevant in the context of this sharing framework. A “finished product” should attempt to address most of these topics, and ideally include tools and services for managing resources and orchestrating workloads on the cluster level. However, we consider this to be outside the scope of this dissertation, and we only focus on the fundamental mechanisms that enable the low-level sharing functionality.

1.4 Research methodology

Choosing an appropriate research methodology for problems in computer science is challenging. Many methodologies come with their own set of considerations and potential pitfalls [33]. Finding a methodology is not made easier by the fact that computer scientists themselves do not seem to agree on the age-old philosophical question of whether computer science should be considered applied mathematics, engineering, or a science [10].

According to Eden [13], computer science as a discipline can broadly be divided into three different research paradigms:

- The rationalist paradigm, which defines computer science as a branch of mathematics. The paradigm seeks certain, a priori knowledge of systems or processes through means of deductive reasoning.

- The technocratic paradigm, which defines computer science as an engineering discipline. The paradigm seeks probable, a posteriori knowledge about systems or processes through implementation (or prototyping) and empirical validation in the form of testing.

- The scientific paradigm, which defines computer science as a natural (empirical) science. The paradigm seeks both a priori and a posteriori knowledge about systems or processes by combining formal deduction and scientific experimentation.

Note that the technocratic concept of a "test" differs from scientific experiments, in that the purpose of the former is to establish to which extent a set of requirements are met, whereas the latter is designed to corroborate or refute a hypothesis. If a test fails (to meet a requirement), the implementation must be revised. If an scientific experiment "fails", it is the theory (or understanding) that must be revised instead.

An almost identical classification of paradigms is given by the ACM Task Force on the Core of Computer Science [9].[2] They additionally note that the three paradigms are intrinsically intertwined, as computer science is both deeply rooted in mathematics and has its own theory, experimental method, and engineering—in contrast to most physical sciences, where the engineering disciplines that apply their findings are considered separate disciplines. The three paradigms are therefore equally fundamental to computer science.

The subject matter of this dissertation touches on several sub-areas of computer science, including computer and hardware architecture, distributed computing, OS fundamentals, and communication networks. While all three paradigms can be applied to these sub-areas [9], it is the technocratic paradigm which lends itself best to answering the overall problem statement of this dissertation (Section 1.2). Neither the rationalist nor the scientific paradigm are particularly well-suited. It is unrealistic to make a model through a priori knowledge alone, due to the complexity of the many different hardware and software components of real-world computer systems. Similarly, the process of gathering data with the goal of understanding the behavior of an indeterministic system, in order to create a statistical model and make predictions about it, seems equally unfit in this context.

As many disaggregation solutions already exist, the motivation for this dissertation is not simply to make it easier for machines to share resources efficiently, but to do so by using a new approach altogether: we attempt to unify traditional device I/O with distributed, shared-memory computing by utilizing the inherent memory mapping capabilities of NTBs. It is desirable to realize this approach by **building a working prototype**, in order to explore potential opportunities and weaknesses along the way. As such, we have followed the

[2]ACM use the names "theory", "design", and "abstraction" for these paradigms instead.

technocratic paradigm by iteratively designing, implementing, and testing our solution according to the objectives given in Section 1.2:

- One of the mechanisms developed makes remote resources appear and behave exactly as if they were part of the local PCIe tree. In order to not require any special adaptions of existing hardware, device drivers, or application software, this mechanism is completely transparent. This mechanism must is thoroughly tested using a wide range of functionality tests, for many different types of devices and device driver implementations. As we make a point of using unmodified hardware and software, commodity devices and widely available software, such as standard benchmarking tools and well-known applications, is used in the validation of our solution. We also include tests using a variety of different cluster network topologies and machine configurations, including VMs, to provide a complete functionality evaluation.

- A wide range of latency and throughput benchmarks is used to measure the performance of the critical I/O data path. Since the solution allows machines to use the internal devices and memory in other (remote) machines as if these resources were locally installed, it is possible to rely on comparison testing. Tests looking at individual I/O operations, such as reading/writing to an NVMe or copying memory out of GPU memory, can first be performed using local resources (without our solution) to establish a performance baseline. Then, the tests can repeated for remote resources using our solution, allowing us to compare the results and subsequently identify which component of the solution that needs to be improved.

- Our implementation process should also identify and explore new possibilities that are enabled by the solution. The strength of our shared-memory approach is best highlighted by demonstrating capabilities that other resource sharing solutions lack. For instance, as CUDA unified memory [47] and GPUDirect [37, 46] can be supported by our solution, even for GPUs that reside in *different machines*, we have performed experiments with direct memory-to-memory transfers across the cluster network. Other possibilities, such as exploiting memory disaggregation to implement memory locality optimizations or using PCIe multicasting to replicate data across several machines in a single operation, is also explored. We investigate not only how different cluster network topologies affect the data path, but also prove the flexibility of our solution and demonstrate several of the sharing scenarios that are made possible.

- Realistic and I/O-intensive computing tasks, e.g., machine learning and image processing workloads, are used to put the solution under heavy load. By running real-world applications using several I/O resources, any accumulated effects of any performance overhead caused by the implementation that are not visible on their own should be revealed. Moreover, this kind of stress testing also proves that the solution is stable and gives reliable performance for repeated runs, and that it does not have any unintentional side-effects that affect systems over time. Finally, showing that the solution works for a realistic workload has the additional purpose of proving that scaling real-world applications is possible.

For the sake of convenience, functionality testing and the overall validation process is implicit in the performance experiments presented in this dissertation. However, it should be mentioned that while the presented experiments primarily use Intel Xeon as the CPU architecture, and NVMes, Ethernet NICs, and Nvidia GPUs were used for shared resources, additional CPU architectures and devices were also used during the development and validation phases. This includes CPU architectures like AMD and ARM, and other devices like FPGAs, AMD GPUs, sound cards, and PCIe-attached cameras.

1.5 Contributions

This dissertation contributes to the topic of resource sharing and distributed I/O facilitation in cluster computing systems, and has been presented in five peer-reviewed venues: two conference workshop publications, one short-length demonstration paper, and two journal articles. These publications are included as Papers I to V and contain the bulk of the implementation details, particularly Paper V, which presents the entire solution as a whole.

We have developed a framework called *SmartIO* for sharing resources and distributing devices in a heterogeneous, PCIe-networked cluster. In particular, the main contributions of this dissertation are listed as follows:

- Implementation of the *Device Lending* method for **distributing PCIe devices to remote systems** (see Paper I): using Device Lending, any standard PCIe device, such as NVMes, GPUs, NICs, and FPGAs, may be assigned to a remote system. The device appears to the remote system as if it has been dynamically hot-added to the system, allowing existing device drivers to use the device without requiring any modifications to software.

- Implementation of a **new method for distributing devices to VMs** running on any host machine in the cluster (see Papers III and IV): we have developed an extension to the KVM hypervisor based on the *Mediated Device Driver (MDEV)* interface, enabling direct access to remote physical hardware devices for VM guests and setting up memory mappings for the devices. This MDEV implementation includes a method for dynamically discovering guest-physical memory layout. Using this MDEV extension, local and remote devices can be "passed through" to VMs and used with bare-metal performance.

- Improvement of the Device Lending and MDEV methods by implementing **support for multiple devices** and **supporting devices in different physical machines** (see Papers III and IV): a method for resolving device memory addresses and setting up memory mappings, in a way that is transparent to both the devices and the device drivers, is implemented. This enables direct data transfers between multiple devices without violating the principle of making devices appear local to the system(s) using them.

- Extension of the SISCI shared-memory API with new, **device-oriented programming semantics and support functions for writing device drivers as shared-memory applications** (see Paper V): this API extension makes it possible to disaggregate devices and device memory in software, similarly to RDMA disaggregation solutions. Unlike RDMA solutions, however, remote resources can be memory-mapped directly into the virtual address space of a software process. Through our API extension, device driver implementations may take full advantage of PCIe shared memory capabilities, such as remote memory access and multicasting, without requiring awareness of the underlying PCIe topology and the different address domains of remote systems. This makes it easier to optimize data flow through the PCIe network, as software no longer needs to be written with accessing remote resources in mind, but can be implemented as if resources are local.

- Development of a **new distributed NVMe device driver**[3] using our device-oriented API extension (see Paper V): although the Device Lending and MDEV methods make it possible to use existing device drivers, most device drivers are written in a way that assumes exclusive control over the device. Therefore, a distributed device (function) may only be used by a single user at the time. To demonstrate software-enabled disaggregation, we have implemented an NVMe driver as a user space SISCI application. As a proof of concept, we show that a single NVMe device can be shared and operated by multiple cluster machines simultaneously, without requiring SR-IOV. This driver also demonstrates how multiple sharing aspects of SmartIO may be combined, by disaggregating remote GPU memory and enabling memory access optimizations, supporting writing to and reading from storage directly to GPU memory. The NVMe driver may even run as a CUDA application on a remote GPU, allowing GPU threads to read and write to storage without involving the CPU.

- A **comprehensive performance evaluation** covering all parts of SmartIO and the implementation of performance optimizations (see Papers II, IV and V): with the goal of not incurring any performance overhead beyond that of native PCIe, the performance of using remote resources with SmartIO is comparable to that of local access (in terms of latency and throughput). To prove that SmartIO is a viable and efficient solution for I/O resource sharing also for realistic scenarios, two different

[3]The prototype NVMe device driver is open source and can be found at https://github.com/enfiskutensykkel/ssd-gpu-dma . It has also been adapted and used in other research projects: https://github.com/ZaidQureshi/gpudirect-nvme [43].

image classification workloads relying on multiple GPUs and NVMe storage have also been tested.

Finally, it should be noted that the research of this dissertation has had impact on real systems, as several components of SmartIO have already been incorporated into the product line of Dolphin Interconnect Solutions, and others are currently being adapted for real-world deployment.[4]

1.6 Outline

This dissertation describes the design, implementation, and evaluation of the SmartIO framework for efficient sharing of resources between PCIe-networked computers. The complete framework is described in detail in Paper V, with the individual components and parts explained in Papers I, III and IV. These papers also show the evolution towards the finished solution, including the gradual performance improvements of the different iterations. The rest of this dissertation is organized as follows:

Chapter 2 gives a summary of SmartIO. We describe the initial idea and challenges for SmartIO, and provide an overview of the implementation. We also present an overview of related work focused on I/O resource sharing in cluster networks.

Chapter 3 summarizes our work and presents ideas for future work.

Paper I describes the initial implementation of the Device Lending component of SmartIO. We evaluate how performance is improved by using Device Lending to enable DMA transfers directly between a GPU and remote memory, compared to using RDMA to achieve the same.

Paper II is a demonstration of how the Device Lending component can be used for a realistic video stream processing workload implemented for GPUs. We demonstrate how real-time processing requirements can be met by using Device Lending to scale up the number of available GPUs.

Paper III presents the MDEV component of SmartIO and how we improved the initial Device Lending implementation with support for multiple devices residing in different machines. We evaluate how performance is improved by enabling peer-to-peer DMA transfers directly between devices instead of needing to bounce data transfers via system memory, and also identify performance issues with relying on the IOMMU.

Paper IV is an extension of Paper III and provides an extended evaluation of both MDEV and Device Lending. We show how different cluster network topologies affect DMA performance, for both host machines and VM guests. Additionally, we also describe how we improved the original

[4] https://www.dolphinics.com/solutions/pcie_smart_io.html

MDEV component with a mechanism for detecting guest-physical memory layout.

Paper V presents the entire SmartIO solution as a whole, with the final iteration of the Device Lending and MDEV components. We also introduce the SISCI API extension that makes it possible to disaggregate devices and memory resources in software, and a proof-of-concept device driver implementation that uses this extension. A thorough and complete performance evaluation of *all* parts and components of SmartIO is provided, in order to prove that the final implementation does not cause *any* performance overhead compared to native PCIe.

Chapter 2

SmartIO

SmartIO is a solution for allowing the local resources of a machine, i.e., memory and devices, to be shared with and used by remote machines, over standard PCIe. Our solution works for *all* standard PCIe devices and their Linux device drivers, no special adaptation is needed in either hardware or software to make this sharing possible. SmartIO is fully distributed and avoids dedicated servers. All machines in the cluster network may contribute their own local resources and access remote resources. Furthermore, as remote devices and memory resources are accessed over native PCIe, they can be shared and used by remote machines with very low latency and extremely low computing overhead. Whether devices are actually local or remote becomes irrelevant to the user, as SmartIO eliminates this distinction, with regard to both functionality and performance.

2.1 Underlying idea

The defining feature of PCIe [42] is that devices are mapped into the same address space as the CPU and random access memory (RAM), as seen in Figure 2.1. This allows a CPU to read from and write to device memory in the same manner

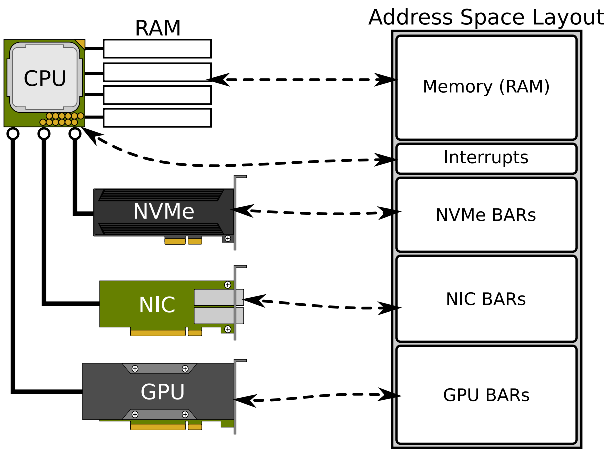


Figure 2.1: Device memory regions (BARs) are mapped to the same address space as CPU and system memory, allowing the CPU to read from and write to device memory the same way it would access RAM. Devices can similarly use DMA to read from and write to RAM.

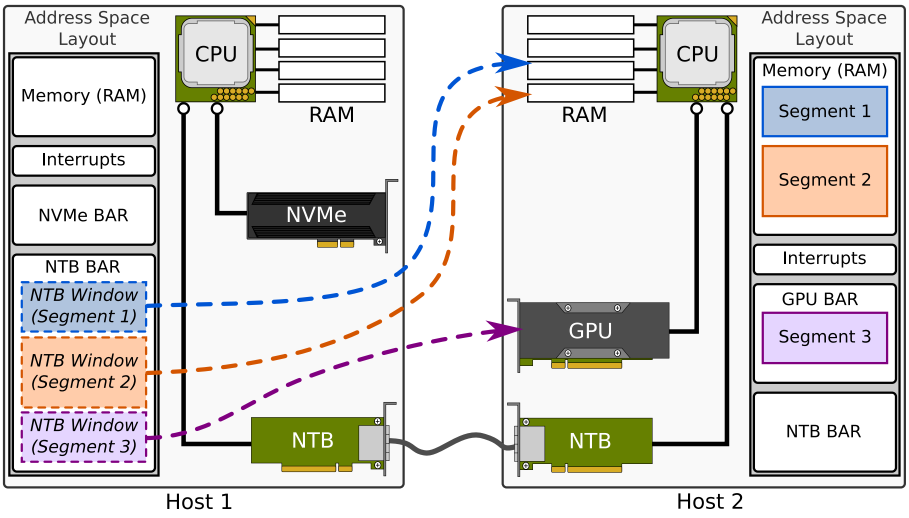


Figure 2.2: Two computer systems connected together using NTBs and external cables. Host 1 has mapped segments of Host 2's memory through its local NTB, providing Host 1 with "windows" into the remote system's address space. The NTBs translate addresses between the two independent address spaces.

it would access RAM, also known as memory-mapped I/O (MMIO). Likewise, devices capable of DMA may read from and write to RAM directly. PCIe also uses message-signaled interrupts (MSI), allowing devices to raise interrupts by writing to an address reserved by the CPU instead of requiring dedicated interrupt lines.

This address space mapping occurs when a system enumerates the PCIe bus and accesses the configuration space of each device. A configuration space contains a description of the capabilities of a device, such as its memory regions. The system will reserve a memory address range for each of these device memory regions, and by writing the start address of these regions to the device's Base Address Registers (BARs), a device is made aware of the address space mapping. Therefore, the term "BAR" is used interchangeably for a region of device memory. Addresses reserved by the system for interrupts are also written to the device's configuration space. For more details on PCIe, particularly how memory transactions are routed, please refer to Section 3 of Paper V (on page 149) and Section 2 of Paper IV (on page 114).

As depicted in Figure 2.2, it is possible to connect computer systems with different address spaces together over PCIe by using NTBs. NTBs can be embedded as a CPU feature [50, 66], but are more commonly implemented in PCIe switch chips [5, 6]. By using such NTB-capable switch chips to implement peripheral devices, independent computer systems can connect with plug-in host adapter cards and external cables. To the system, the NTB appears as a normal PCIe device[1] with one or more BARs that are reserved and mapped during the enumeration process. However, rather than being backed by device registers or device memory, the NTB instead forwards reads and writes to its BARs from one side of the NTB to the other, translating memory addresses in the process. The NTB uses a look-up table for address translation, which can be configured dynamically during run-time. By using different base offsets in this look-up table, it is possible to configure several memory-mappings (or "windows") into the address space of a remote system. Figure 2.2 illustrates how arbitrary memory addresses on the remote system can be mapped, allowing the local CPU to access remote memory as if it was local device memory. Although address translation between the different address spaces is very fast since the look-up table is implemented in NTB hardware, the number of NTB windows is limited by the maximum number of table entries. More details on how NTBs work can be found in Section 3.3 of Paper V (on page 151).

Since device memory on a remote system is part of the same address space as system memory, we can use an NTB to map memory of a remote device. We show this in Figure 2.2, where Segment 3 is allocated in GPU memory rather than system RAM, but still mapped for the CPU of Host 1's similarly to the other segments. By mapping all BARs of a remote device for a local CPU, it would be possible to perform memory operations on the remote device, such as reading from or writing to device registers. Moreover, device DMA is not limited to reading and writing to system RAM, but can also be used to access memory on other devices in the same address space. This is known as "peer-to-peer" in PCIe, and provides us with an opportunity as it becomes possible for a device to read and write directly across an NTB. We can use this to map memory resources for a device, be it RAM or memory of other devices. Furthermore, because PCIe uses MSI, it is even possible to map interrupt addresses through an NTB, as they too are mappable.

2.2 Main challenges

Although NTBs provide the fundamental memory mapping capabilities that can facilitate the use of remote devices, the challenge is to avoid requiring that device drivers must be aware of remote-side address spaces. As touched upon in Section 1.1, this is desirable in order to use existing device driver implementations. For a device driver running on a local machine to be able to use a remote device, we must make sure that the driver uses addresses that is mapped through both the local and remote NTBs. For instance, when the device driver attempts to access device BARs, we must make sure that the driver uses memory addresses that are mapped through the CPU-side NTB without the driver or device being aware of this. Conversely, when the device driver attempts to initiate DMA transfers or configures an interrupt vector address, we must

[1]The PCIe terminology for individual device functions is "endpoints". We use the terms "device" and "function" as synonyms for a PCIe endpoint throughout this dissertation.

find a way to transparently inject memory addresses that are mapped through the remote, or device-side, NTB.

One possibility is to use virtualization to mitigate the complexity of managing different address spaces. The fact that devices are on the other side of an NTB could be hidden for device drivers by distributing devices to VM guests instead of physical host machines, for example with pass-through. However, while pass-through allows devices to be used by VM guests directly, *requiring* that compute tasks run in VMs will limit the generality of a solution. Virtualization is not necessarily appropriate in all circumstances, as CPU cycles are spent on hosting the virtualized environment, thus adding additional system load. Instead, a more general mechanism is needed for abstracting away the complexity of dealing with a remote-side address space. This mechanism must support abstracting remote address spaces for VMs and bare-metal machines alike.

DMA is particularly challenging in this context. A device driver running on the host machine may assume that any local memory address can be reached by the device, but as explained in Section 2.1, the NTB only provide *windows* into a remote address space. It is generally not possible to predict in advance which memory addresses a device driver may use, yet memory must be mapped through the NTB before the driver, unaware that the device is remote, initiates DMA transfers. Deferring the action of mapping memory through the NTB until a device driver initiates DMA, or some other time when the specific addresses of DMA buffers and VM memory can be known, is not viable; synchronizing with the remote system at this time will introduce communication overhead in the performance-critical path. The naive solution is to map the entire system memory for the device, but this workaround requires the NTB BARs to be at least *as large* as the size of RAM. This does not scale very well, as it would effectively limit the number of machines the cluster can support. Each new machine using a device would increase the device-side NTB memory requirements by its entire RAM size. Moreover, as the number of maps supported by an NTB is also limited (by the size of its look-up table), it is crucial to conserve memory maps wherever possible. We must find a way to prepare memory-maps through the NTB in advance of use, in order to avoid adding communication latency in the critical path, and without requiring that the entire memory is mapped for the device.

The challenge of DMA transfers is compounded for VM pass-through. Pass-through is possible by using the IOMMU to create a virtual I/O address space for a device that corresponds to the virtual address space of the VM [29, 30, 36], also known as the "guest-physical address space". For pass-through of a local device, this makes it possible for the device driver running in the VM guest to use any (virtualized) memory address when initiating DMA transfers. In our case, however, the driver must use addresses that are mapped through the NTB in order for a remote device to reach local host-physical memory. In order to support pass-through, we must devise a method for mapping the physical memory backing the VM emulator through the NTB, using device-side I/O addresses that mirrors the guest-physical address space.

The published papers provide further details on the challenges facing our solution as part of the description of the implementation of the different components of SmartIO. Particularly Sections 4 to 6 of Paper V (on pages 153, 156 and 161) provide a more in-depth explanation of what the main challenges are, and they also explain how SmartIO tackle them. A discussion of additional considerations is given in Section 8 of Paper V (on page 200).

2.3 Implementation

In our framework, computer systems act as *"borrowers"* and *"lenders"*. A lender is a computer system that registers one or more of its internal PCIe devices with SmartIO, allowing these devices to be distributed to and used by remote machines. A borrower is a system that is currently using such a device. While a device only has one lender, namely the computer system where it is physically installed, there can be several borrowers using it simultaneously. SmartIO also makes it possible for a system to act as both lender and borrower at the same time, lending out its own local devices and simultaneously borrowing remote devices from other machines.

Basing our framework on standard PCIe is a deliberate design choice. Not only does this allow commodity devices to be operated remotely by standard device drivers over native PCIe, but this design also means that the implementation complexity of SmartIO lies entirely in software. In fact, SmartIO can be implemented for existing computer systems that are connected using NTBs in any network topology, regardless of whether the NTBs are switch chips soldered onto a motherboard or implemented as plug-in adapter cards.

Unlike other solutions for distributed I/O, SmartIO combines traditional I/O with distributed shared-memory functionality in a seamless manner. Sharing is supported at multiple levels: devices may be distributed to physical host machines and to VMs alike. Individual device functions of multi-function devices may be distributed to different machines in the network, or to the same machine should it require multiple resources. SmartIO also provides software facilities for disaggregating devices and memory resources, allowing device drivers to be implemented as part of distributed cluster applications or other user space applications. This makes it possible for several machines to simultaneously share a single device (function). It is even possible to *combine* the sharing methods of SmartIO. For example, we can disaggregate the device memory of a remote GPU using the API extension while it is being borrowed through Device Lending, or we can share VFs of an SR-IOV NIC with both physical host machines and VMs.

In this section we address the challenges of Section 2.2, and give a bottom-up summary of the implementation of SmartIO. Figure 2.3 illustrates the different components of our framework, and how they fit together. The implementation is described in full in the published papers, with Paper V providing a detailed description of the entire solution as a whole.

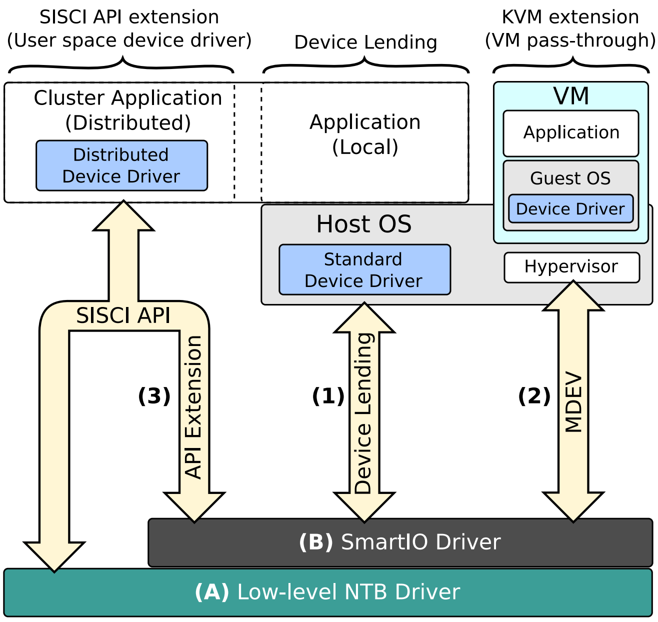


Figure 2.3: The software architecture of SmartIO. Three different sharing methods are made possible by our framework: **(1)** Device Lending, **(2)** MDEV, and **(3)** using the SISCI API extension. The SmartIO driver, shown as layer **(B)**, abstracts away the physical location of remote resources, for both the shared device and software using the device.

2.3.1 Low-level NTB driver

SmartIO is implemented on top of the NTB interconnection solution from Dolphin. The low-level NTB driver, illustrated as layer **(A)** in Figure 2.3, provides the fundamental PCIe networking infrastructure and memory mapping functionality which SmartIO builds on. Individual systems may contribute parts, or *segments*, of their local memory to a distributed, shared memory space. Memory segments in remote machines may be mapped into the local address space of a system by using the NTB, as explained in Section 2.1. Moreover, user space applications may use the SISCI API to interact with the NTB driver to manage memory segments and implement shared-memory communication.

2.3.2 SmartIO driver

The SmartIO driver, shown as layer **(B)** in Figure 2.3, runs on all machines in the cluster. It acts as an abstraction layer, providing a logical decoupling of devices and which physical machines they are installed in (lenders). Neither devices nor software need to consider where resources physically reside, since SmartIO resolves this on behalf of both devices and machines using them (borrowers).

By providing this abstraction, the SmartIO driver is the first step towards the sharing methods presented in Sections 2.3.3 to 2.3.5, namely Device Lending, MDEV, and the SISCI API extension.

Devices registered with SmartIO are assigned a unique identifier which allows machines to refer to a device without needing to specify the lender machine. Internally, the SmartIO driver keeps track of devices and lenders, and uses this device identifier to look up devices and machines, and which NTBs to use. The SmartIO driver is also responsible for making device BARs available as shared memory segments, making it possible for borrower machines to memory map remote device memory into their local address space (MMIO).

Most important is the SmartIO driver's responsibility of mapping memory segments **on behalf of a device** and returning the I/O addresses to these maps, **as seen by the device**. The SmartIO driver works out the physical locations of devices and memory segments, i.e., which machines they reside in and which NTBs a device must use in order to reach a segment. Note that a segment can reside in memory of the machine using the device (the borrower), in memory of the machine where the device is installed (the lender), or a different cluster machine altogether. A segment can even be allocated in device memory of another device, as the SmartIO driver can assist in mapping device BARs and enabling peer-to-peer between devices. Borrowers are not required to know anything about the *device-side* I/O addresses returned by our SmartIO driver, other than the fact that they resolve to the same address space as the device. This allows both a borrower and the device to remain agnostic about the underlying, physical PCIe topology, as they can rely on the SmartIO driver to resolve paths in the cluster network and map resources through the appropriate NTBs.

The SmartIO driver solves the challenge of managing multiple address spaces, as described in Section 2.2. For example, a borrower can request a memory segment in a different machine is mapped so that the device may use DMA to it. Our SmartIO driver will look up which machine the memory segment is in, look up the lender machine and which (device-side) NTB it must use, configure the NTB, map the memory segment for the device, and return the device-side I/O address of this map back to the borrower. The borrower can then use this I/O address when interacting with the device in order to initiate the DMA transfer, and the device is able to reach the memory segment through the lender's NTB. Borrowers do not need to be aware of the internal I/O address space layout of a lender. As it happens, borrowers do not even need to know which physical machine the lender is.

With the abstraction the SmartIO driver provides, our framework is able to facilitate the sharing and use of remote resources (both memory and devices) as described in Sections 2.3.3 to 2.3.5. More details on how SmartIO resolves the paths between devices and other memory resources that must be mapped (for devices) are given in the papers, particularly in Paper V (on page 140). However, please note that the SmartIO driver is not mentioned explicitly by name in the papers, as it is the unifying base for the sharing methods. Instead, the description of its functionality is interleaved with the implementation details of these methods.

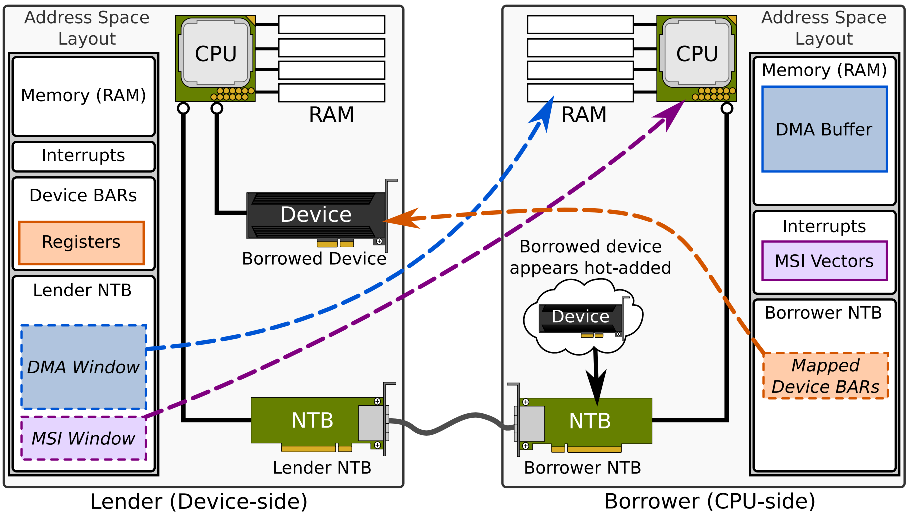


Figure 2.4: The memory regions of a remote device is mapped for the CPU on the borrower, so that it can read and write to device registers. Local resources are mapped for the device, so that it may use DMA and trigger interrupts. Device Lending inserts a shadow device into the local device tree using these mappings, making remote device access transparent to both CPU and device.

2.3.3 Device Lending

Device Lending, illustrated as arrow **(1)** in Figure 2.3, makes it possible to share and distribute devices to remote host machines. The devices become part of the system they are shared with, allowing application software, device drivers, and even the OS to use them as if they were locally installed. While Device Lending only allows individual device functions to be distributed to a single machine at the time, it is nevertheless highly suitable in the case where a device has a complex or proprietary device driver as no modifications to existing software is required.

As mentioned in Section 2.1, it is possible to map the device memory regions, or BARs, of a remote device through an NTB. Using the NTB, a local CPU can perform memory operations on a remote device, such as reading from and writing to device registers. Conversely, local resources, such as RAM and interrupt addresses, can in turn be mapped for the remote device itself. This allows the remote device to use DMA through the NTB and trigger interrupt routines on the local CPU. The SmartIO driver, as explained in Section 2.3.2, eliminates the complexity of managing multiple address spaces: a user may rely on the SmartIO driver to map resources through the appropriate NTBs and provide I/O addresses corresponding to a device's address space. However, as pointed out in Section 2.2, we still need to make sure that device drivers use this functionality without requiring that they be re-written. More precisely, we need a mechanism for *transparently* injecting resolved I/O addresses, without

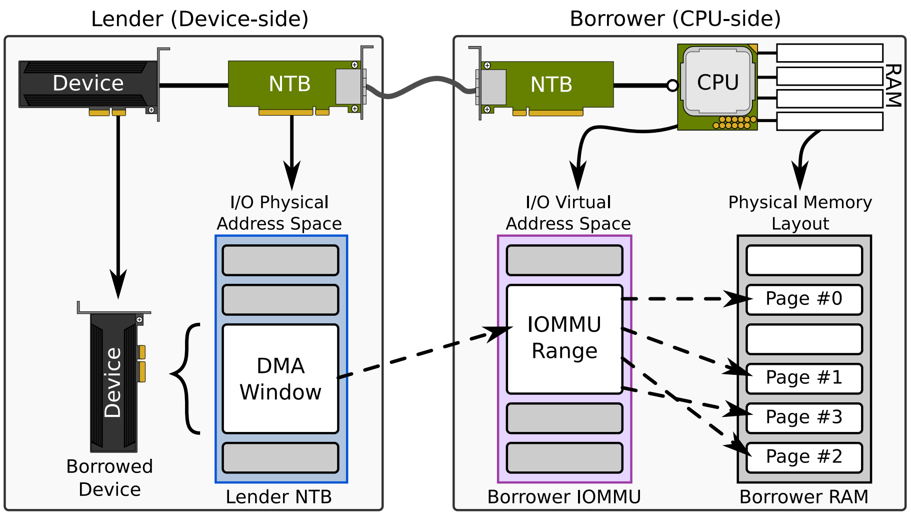


Figure 2.5: The borrower's IOMMU is used to create a single continuous memory range that may be mapped through the lender's NTB in advance. Adding and removing memory pages from the local IOMMU domain is inexpensive compared to actively communicating with the remote lender machine.

the devices or their drivers being aware of this.

Device Lending solves this by inserting a "shadow device" into the local PCIe device tree on the borrower, as depicted in Figure 2.4. The shadow device makes it appear as if the remote device has been hot-added to the local system, and provides us with a mechanism for intercepting interactions with the device by the OS and any device drivers. We make sure that a device driver attempting to memory map the device's BARs use addresses that map through the NTB, without the driver being aware that the device is actually remote. Similarly, when the device driver configures interrupts, we are able to intercept this and inject an address that map through the lender's NTB, again without the driver being aware.

The shadow device also provides us with the means to detect when a device driver is allocating DMA buffers and making memory available for DMA transfers. Unlike device BARs and MSI addresses, memory addresses for DMA buffers are not known in advance, as mentioned in Section 2.2. Mapping BARs and interrupts is a one-time operation. The pages used for DMA memory buffers, however, may be scattered in physical memory. A driver may even initiate several transfers to different parts of memory altogether. Mapping individual memory pages through the lender's NTB would not only exhaust the number of available entries in its look-up table, but communicating with the lender machine in order to map these pages dynamically would introduce communication latency in the critical performance path.

To solve this, Device Lending relies on the IOMMU on the borrower, as depicted in Figure 2.5. We can prepare a continuous memory address range *in advance* using the borrower's IOMMU. This range can be mapped through the lender's NTB with a single map, or "DMA window", even before any device drivers are using the device. When a device driver, at a later point, allocates DMA buffers, we can simply add these addresses to the IOMMU range. This way, we can avoid making any assumptions about the memory used by a device driver. Additionally, since this is an entirely local operation (on the borrower), communication with the remote lender machine is avoided. All address translations between the different address domains are done in NTB and IOMMU hardware, ensuring that this solution is able to achieve native PCIe performance in the data path.

A more technical description of the implementation of Device Lending can be found in Section 4 of Paper V (on page 153), including details about the shadow device and configuration cycles. Section 4.4 of Paper V (on page 155) and Section 6 of Paper IV (on page 120) explain how Device Lending is able to support peer-to-peer DMA between multiple borrowed devices, even when they reside in different lender machines. In addition, Sections 4.3 and 8.4 of Paper V (on pages 154 and 202) explain how we can use the *lender's* IOMMU to remap DMA windows from high to low memory addresses for devices with addressing limitations.

2.3.4 MDEV

Our MDEV implementation, shown as arrow **(2)** in Figure 2.3, enables the Linux KVM hypervisor to *pass through* borrowed devices to VMs. This pass-through allows software running in a VM guest to use the physical device directly, without compromising the memory isolation provided by the virtualization. Whereas Device Lending is only supported for machines running Linux, MDEV makes it possible to share devices with *any* guest OS (running in the VM). Using MDEV, devices are no longer tightly coupled with the host machines they are installed in, allowing VMs to be distributed across different hosts in the cluster while benefiting from the bare-metal performance of direct access to physical hardware. VMs can be migrated to any host in the cluster, as devices are assigned to VMs using device identifiers and dynamically borrowed and returned on boot and shutdown.

Our MDEV sharing method is implemented using the mediated device driver paravirtualization interface [22]. Comparable to the shadow device used by Device Lending (Section 2.3.3), this interface makes it possible to trap (handle) certain operations, such as configuration cycles and device resets, and to set the memory addresses of the device's BARs. However, unlike Device Lending, we do not have any means of controlling which I/O addresses a device driver should use when initiating DMA transfers. In Device Lending, we create a single, continuous IOMMU range ahead of time, and map it through the lender's NTB. We are able to detect when a device driver is preparing DMA buffers through the shadow device, and can inject the prepared device-side I/O address of our DMA window. In contrast, a device driver running in the VM guest is completely

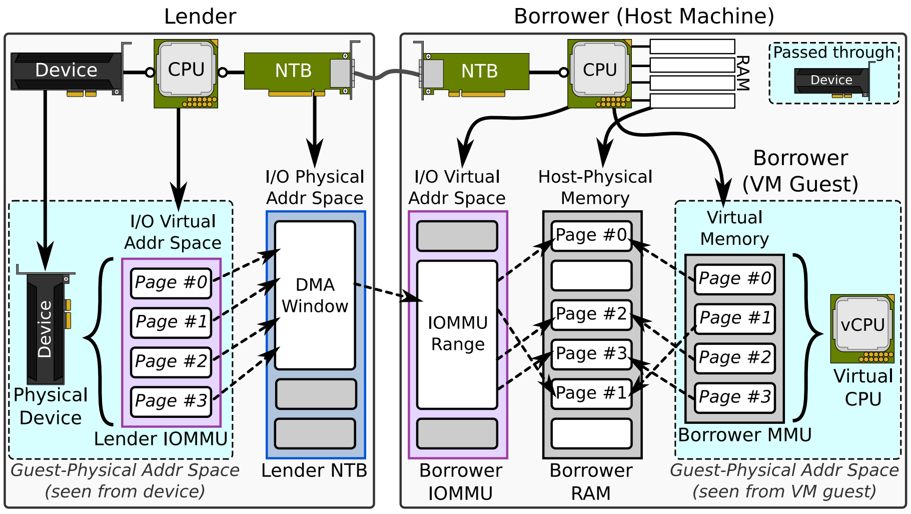


Figure 2.6: The IOMMUs on both sides of the NTB must be used in order to pass through a remote device to a local VM. The borrower's IOMMU is used to provide continuous memory ranges for scattered VM memory, while the lender's IOMMU is used to mirror the guest-physical layout for the device.

isolated, leaving us without any equivalent mechanism to inject I/O addresses we can control. The only possible option is to make sure that the device is mapped to the same virtual address space as the VM, as pointed out in Section 2.2.

In order for a device to DMA to VM memory, the host-physical memory pages backing the emulated memory needs to be locked in physical RAM. In practice, all memory allocated for a VM guest must be mapped for the device, as a device driver or application running in the guest may try to use any guest-physical address when interacting with the device. However, as this is handled by the hypervisor for normal pass-through of a local device, the mediated device driver interface does not provide any notification of when this memory is allocated by a VM emulator. As such, we have no reliable method of detecting host-physical memory that needs to be mapped. To further complicate matters, we cannot rely on modifying existing emulator software to solve this issue, as per our objectives stated in Section 1.2. Nevertheless, it is possible to rely on an assumption: before a device can use DMA, it must be enabled in its configuration space.[2] Since the mediated device driver interface makes it possible to trap configuration cycles, our MDEV implementation waits for DMA to be enabled. By then, the emulator has allocated all of the host memory it needs for the VM, and we are able to use the hypervisor to lock the host-physical pages of the VM in physical RAM and resolve their physical addresses.

With the host-physical memory backing the VM resolved, we can now map it for the device. However, when passing through a local device, the

[2]Enabling the “Bus Master” bit in the command register enables DMA for a device.

hypervisor places the device in a IOMMU domain with I/O virtual addresses that correspond to the VM's address layout. In our case, the device resides in a *different* machine, i.e., the lender. Additionally, the host-physical memory pages used by the emulator may be scattered throughout physical RAM. Our MDEV implementation solves this by using both the IOMMU on the borrower and on the lender, as shown in Figure 2.6. The borrower's IOMMU is used to provide continuous address ranges that can be mapped through the lender's NTB, or DMA windows. The IOMMU on the lender is used to map the device to a virtual I/O address space that mirrors the VM guest's, allowing a device driver running in the VM guest to initiate DMA transfers using guest-physical addresses.

Section 5 of Paper V (on page 156) provides more information about the MDEV implementation, including a more detailed description of the mediated device driver interface and the functionality it provides. In the same section, we also explain a workaround for interrupts, by relaying them from the lender to the borrower. A method of probing the KVM hypervisor using well-defined starting addresses for high and low memory is outlined, in order to pinpoint the guest-physical memory layout and conserving NTB maps. Finally, we also discuss some security implications of our pass-through approach in Section 8.1 of Paper V (on page 200).

2.3.5 API extension

As mentioned in Section 2.3.1, a user space application may memory-map shared memory segments into its own local address space using the SISCI API. Application processes running on different machines may read and write to remote memory, as if it was reading or writing to local RAM. Our API extension, depicted as arrow **(3)** in Figure 2.3, adds device-oriented programming semantics and device driver support functionality to this shared-memory API. This extension makes the core SmartIO capabilities, as described in Section 2.3.2, available through the same shared-memory API used to write cluster applications.

The BARs of a device is exported as shared memory segments that may be mapped by the application, providing access to device registers and other device memory (MMIO). Several application processes running on different machines may even access such memory mapped BARs at the same time. Similarly, memory segments can be mapped for a device (as DMA windows), allowing devices to use native DMA to access shared memory segment directly. These segments can reside in local RAM of the lender, the borrower, or a different cluster machine entirely. Segments can even be allocated in device memory of *other* devices. SmartIO dynamically resolves the location of segments and devices in the cluster network, and can set up and tear down the necessary NTB maps for the respective machines. The API extension also provides functionality for allocating segments associated with a device, and using access pattern hints in order for SmartIO to determine which machine it should allocate memory in.

By allowing device drivers to be implemented as part of the application software, we make it possible for devices and device operation to become part of the same shared global address space as distributed shared-memory applications. Thus, device drivers can be implemented in a way that fully utilize the capabilities of PCIe networks. For instance, applications may stream data to several destinations in a single operation, replicating data across several machines by relying on multicasting support in the PCIe hardware.[3] A programmer can exploit memory locality to optimize data flow through the network without needing to be aware of the actual network topology, by specifying memory access pattern hints and allowing SmartIO to decide where segments should be allocated. It is even possible to combine the API extension with the other sharing methods of SmartIO, allowing disaggregation of *borrowed* resources. An example of this is would be a machine using Device Lending to borrow a remote GPU capable of GPUDirect [37, 46], allowing the device to be operated by the native driver, and then using the API extension to export GPU memory as a shared memory segment. Application processes on other machines can then memory map this (remote) device memory segment, allowing them to read and write directly to it as if it was local memory.

Since the API extension is built on the underlying SmartIO driver, software can be written in a way that does not need to consider whether resources are local or remote. However, using the API extension requires the implementation of new device driver software. Implementing a driver from scratch may not be a viable option in many cases, as it typically entails a considerable engineering effort. After all, the strength of Device Lending and the MDEV implementation is precisely that they do not require any modifications to existing device driver software. Even so, being able to integrate device drivers into the cluster application itself may potentially be very useful for some application domains. The possibilities created by the API extension are perhaps best exemplified by our proof-of-concept NVMe driver, described in Section 2.3.6. This driver shows how a non-SR-IOV NVMe can be disaggregated in software and shared by several machines, as well as how we can use the API extension for disaggregating memory resources. A more specific description of the functionality added to SISCI by our API extension can be found in Section 6.1 of Paper V (on page 162).

2.3.6 Proof-of-concept NVMe driver

Most device drivers are written in a way that assumes exclusive control over the device. In most cases, a device can only be distributed to a single borrower machine at the time, preventing others from using it while it is used. Some devices implement SR-IOV [41], making a single physical device to appear as multiple (virtual) device functions, or VFs. Using Device Lending or our MDEV implementation, it is possible to distribute such VFs to borrowers. However, due to the complexity of supporting virtualization in hardware, SR-IOV is not widely available, especially not for commodity devices. By facilitating disaggregation in *software* instead, our SISCI API extension (Section 2.3.5) makes it possible for several application processes running on different machines to share the same device (function). As a demonstration of this software-enabled disaggregation,

[3]PCIe multicasting is defined in the PCIe specification [42]

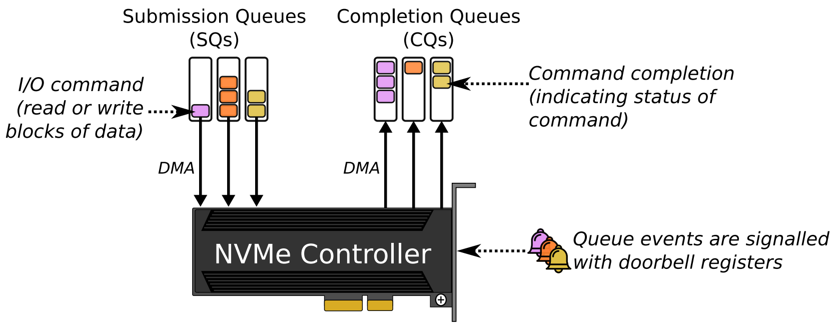

Submission Queues (SQs)
Completion Queues (CQs)
I/O command (read or write blocks of data)
Command completion (indicating status of command)
DMA
DMA
NVMe Controller
Queue events are signalled with doorbell registers

Figure 2.7: NVMes support parallel and asynchronous operation by using independent queues for submitting I/O commands and receiving completions. Queues are hosted in memory, and an NVMe uses DMA to fetch commands and post completions.

we have implemented an NVMe driver as a user space SISCI application. NVMes are highly parallel by design, and the interaction between driver software and NVMe hardware is standardized [14], making it possible to implement a general device driver for them.

Figure 2.7 shows how NVMes support asynchronous operation by using a system of paired command submission queues (SQs) and completion queues (CQs). Both types of queues are data structures that are allocated in memory by discretion of the driver, and may be placed in *any* memory location. The driver submits I/O commands, such as reading or writing N blocks from storage, to an SQ. The NVMe will use DMA to fetch commands from the SQ, and once a command is carried out, the NVMe posts the command completion status to the associated CQ (also using DMA) containing the status of the command. "Doorbell registers" on the NVMe device are used to signal when new commands should be fetched, and each queue has its own doorbell register. Furthermore, as *multiple* of these queue pairs can be created, NVMes avoid any contention in the command submission and completion paths. An example of this is a multi-core CPU that assigns an SQ and an associated CQ per CPU core, allowing each core to operate the NVMe independent of others.

As shown in Figure 2.8, our own driver implementation works by taking this one step further, allowing CPUs in different machines to operate an NVMe simultaneously using their own queue pairs. Each machine allocates a memory segment where it sets up the queues' data structures, and uses the API extension to map these segments for the device as DMA windows. This allows the NVMe to read SQ memory and write to CQ memory the same way it would access local memory, using native DMA. Likewise, as device BARs are automatically exported by SmartIO as segments, all machines can memory map doorbell registers for their respective queues. Since queues are completely parallel, all the machines can submit I/O commands and receive completions entirely independent of each other, once the queues are configured. Note that all

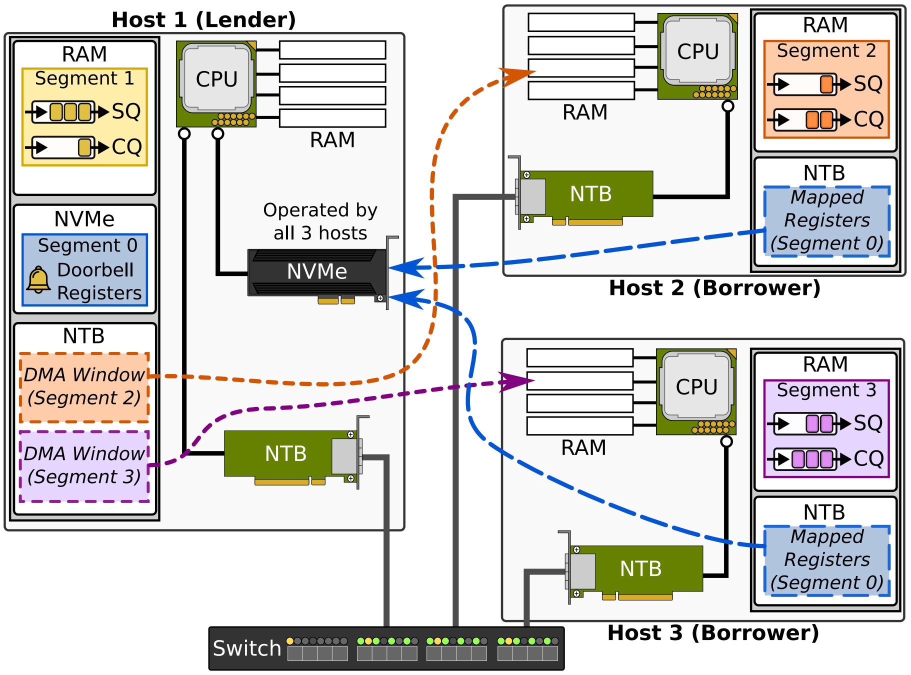
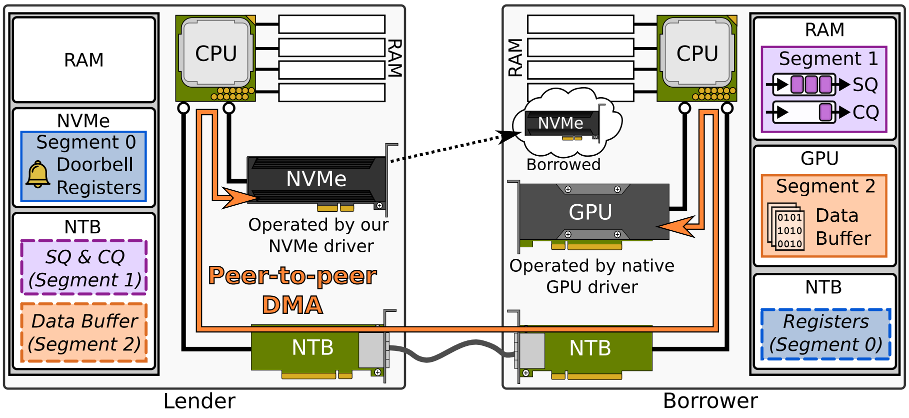


Figure 2.8: Several machines can share and operate the same NVMe simultaneously by distributing queues. Using the SISCI API extension, memory segments with the queues' data structures can be mapped for the device, and doorbell registers can be memory mapped for application processes.

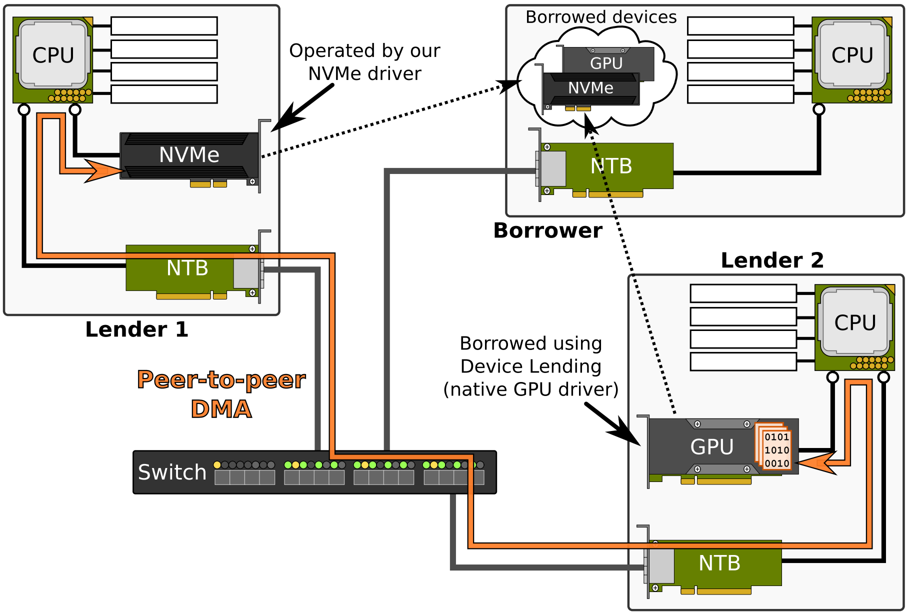
machines are able to operate the NVMe at the same time, including the lender. The software is the same regardless of which machine it runs on, as SmartIO keeps track of where the segments and the device reside.

As mentioned in Section 2.3.5, while using API extension requires developing a new device driver, such as our proof-of-concept NVMe driver, the benefit is that device operation can become part of the cluster application itself. Because SmartIO abstracts away the location of devices and memory segments, the complexity of developing such distributed drivers is somewhat alleviated as software can be written in a way that does not need to consider whether resources are local or remote. Any memory resource can be mapped for the device, regardless of its location in the cluster. An implementation can exploit this in order to optimize the movement of data through the network—without needing to consider the actual PCIe topology. It is even possible to combine the use of the API extension with the other sharing methods of SmartIO. Our NVMe driver demonstrates some of the possibilities created by the API extension:

Remote GPU access (Figure 2.9): Many GPU-accelerated applications, such as big data and machine learning tasks, require access to data on a storage device. Traditionally, loading data from a storage device and



(a) Storing from and loading into the memory of a GPU residing in the borrower.



(b) Storing from and loading into the memory of a borrowed (remote) GPU.

Figure 2.9: By relying on GPUDirect to expose GPU memory through the GPU's BARs, our proof-of-concept NVMe driver is able to map GPU memory for an NVMe using the SmartIO API extension. This makes it possible to load and store GPU data directly, without unnecessarily copying it via RAM.

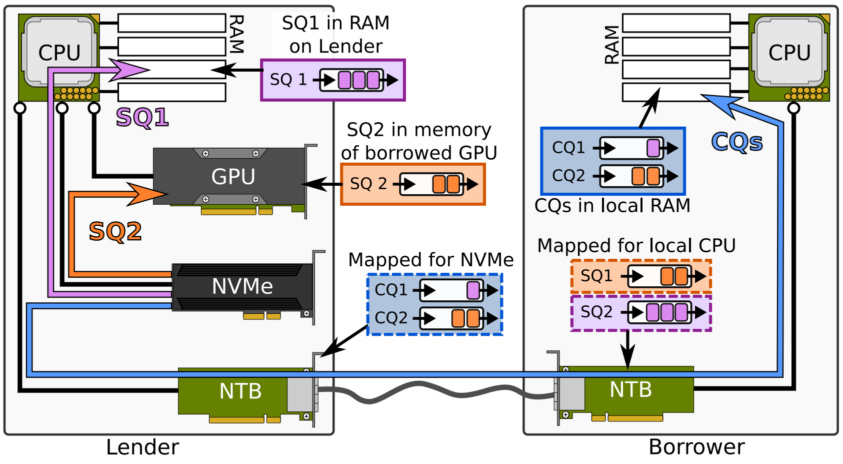


Figure 2.10: Our NVMe driver implementation relies on SmartIO to decide where segments containing queues should be allocated. By using access pattern hinting, it is possible to consider memory locality without requiring the driver implementation to be aware of the underlying PCIe topology.

into GPU memory involves first reading data to system memory, and then copy it onto the GPU. Likewise, storing the result of a GPU computation involves copying it out of the GPU to system memory, and then writing it to storage. Since datasets used in typical big data and machine learning tasks can be as large as hundreds of terrabytes, GPU applications become bounded by transfers between storage and GPU. To overcome this, some GPUs support peer-to-peer DMA, making it possible to load data directly into GPU memory and avoid unnecessary copies via system memory [4, 54]. For Nvidia GPUs, this functionality is supported through the GPUDirect API [37], which makes on-board GPU memory accessible through the GPU's BARs. As explained in Section 2.3.2, SmartIO automatically exports device BARs as memory segments, which can be mapped for a device using the API extension. Our proof-of-concept NVMe driver use this to enable an NVMe to read and write directly to the memory of both local GPUs and *borrowed* GPUs (using Device Lending), as seen in Figures 2.9a and 2.9b respectively. Note that the GPU is operated by the native GPU driver in both scenarios, but we still use the API extension to disaggregate its device memory.

Memory locality optimizations (Figure 2.10): In PCIe, the latency of transactions are affected by the number of switch chips they need to traverse. Particularly memory reads are affected; the longer the path between a device and the memory it reads from, the higher the latency becomes, as PCIe transactions have to travel further. As described in Section 2.3.5, a programmer can rely on the API extension to decide where segments

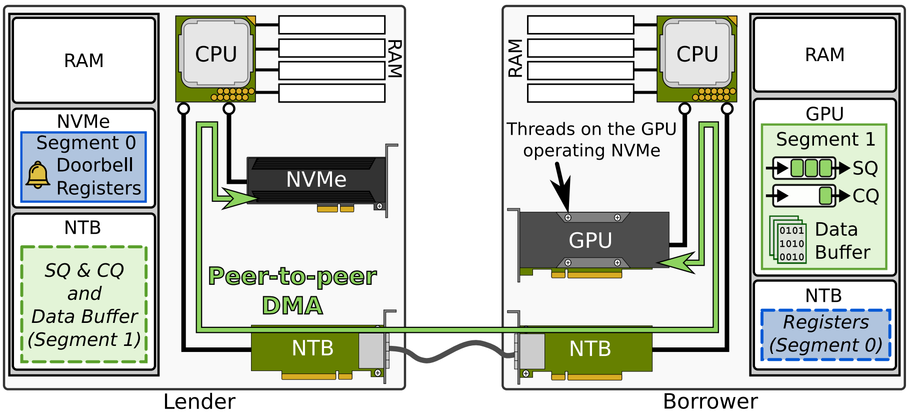


Figure 2.11: Our NVMe driver can also run on a GPU. GPU threads can submit I/O commands and wait for completions independent of the CPU.

should be allocated, based on memory access pattern hinting. In the case of our NVMe driver, we can use this functionality when creating segments for queue memory, as illustrated in Figure 2.10. By specifying that the segment containing CQs will be written to by the NVMe, and read by the borrower's CPU, SmartIO will decide to allocate the segment in memory closer to the CPU, i.e., in the borrower's local RAM. Similarly, a segment with an SQ can be allocated in the lender's RAM by specifying that the NVMe will read from it and the CPU will only write to it, shown as SQ1 in Figure 2.10. It is even possible to use memory of *another device* for SQs. By mapping GPU memory for the device, as explained above, it is possible to allocate the SQ on a borrowed GPU that is close to the NVMe, depicted as SQ2 in Figure 2.10.

Initiating I/O directly from the GPU (Figure 2.11): Similarly to how GPU memory can be mapped for an NVMe using SmartIO, it is also possible to map doorbell registers for the GPU. The logic for submitting I/O commands to an SQ and polling an CQ for completions is also supported for CUDA applications in our NVMe driver implementation. As such, **our NVMe driver may run on a GPU**, allowing GPU threads to read from and write to storage directly without any involvement of the CPU. The performance of CUDA applications that work with large data sets, such as machine learning workloads, can be increased, as loading and storing data at various points in the computation no longer requires synchronization with software running on the CPU.

A more technical description of the implementation of the proof-of-concept NVMe driver can be found in Section 6 of Paper V (on page 161). Here, we describe how the driver is split into a manager and a client component, with the manager being responsible for resetting the device and assigning queues to clients. We also explain how our implementation can support multi-path fail-over, and how PCIe multicasting can be used to replicate data loaded from storage across machines in a single operation. More details on how it is possible to run the NVMe driver as part of a CUDA application can also be found here.

2.4 Performance measurements

Our SmartIO framework makes it possible for machines to use remote PCIe devices in a manner that is indistinguishable from using local resources, both functionally and performance wise. Since remote devices can be used without requiring any modifications to either application software or device drivers, it is possible to use standard benchmarking tools and existing application to evaluate SmartIO. An evaluation of SmartIO in the form of a comprehensive collection of performance tests can be found in the published papers, ranging from microbenchmarks aimed at evaluating specific components to large-scale, realistic workloads:

- In Section 6 of Paper I (on page 84), the initial Device Lending implementation is evaluated using an Nvidia GPU, comparing the performance of DMA transfers of a borrowed (remote) GPU to a local GPU. We also compare the performance of native DMA to a PCIe-based RDMA implementation, to demonstrate the performance benefit of native DMA transfers to remote memory.

- The improved Device Lending implementation with support for peer-to-peer between devices is evaluated in Section 7.3 of Paper III (on page 105), where we perform peer-to-peer experiments with GPUs in several network topologies and test scenarios, demonstrating the performance benefit of peer-to-peer DMA compared to bouncing data via RAM. These tests are extended in Section 7.2 of Paper IV (on page 123) with additional network topologies and test scenarios. We also show the performance a SR-IOV-capable NIC being shared with multiple machines at the same time in Section 7.1.6 of Paper V (on page 181).

- Our MDEV implementation is evaluated in Section 7.2 of Paper III (on page 105), where the performance of a VM using a passed-through is measured. The impact of IOMMUs on DMA transfers is demonstrated in Section 7.3.1 of Paper V (on page 187). In Section 7.3.2 of Paper V (on page 188), we extend our MDEV evaluation and compare the performance of our MDEV implementation to “regular” pass-through (of a local device) as well as bare-metal Device Lending. We prove that although the use of an IOMMU comes with an unavoidable performance penalty, our MDEV implementation does not add any performance overhead compared to bare-metal performance (with the IOMMU enabled). Additional peer-to-peer experiments using MDEV are also presented in Section 7.3 of Paper IV (on page 126).

- To prove that Device Lending and MDEV can be used for real-world applications, we evaluate the run-time performance of a machine learning workload using borrowed GPUs and a borrowed NVMe in Section 7.2 of Paper V (on page 185) and Section 7.5 of Paper IV (on page 130). To further demonstrate SmartIO being used for realistic applications, we present a GPU-based video processing workload in Paper II (on page 89), and use Device Lending to scale up the number of available GPUs for the workload, in order to meet a real-time deadline.
- Several experiments using our our proof-of-concept NVMe driver are presented in Section 7.4 of Paper V (on page 190), demonstrating capabilities made possible through the SISCI API extension. We show how 30 cluster nodes can share a single-function NVMe simultaneously, how data stored on the NVMe can be multicasted and replicated across 60 nodes in a single operation, as well as interoperability with both local and remote GPUs. A comparison of our proof-of-concept NVMe driver and a state of the art NVMe-oF solution is presented in Section 7.4.3 of Paper V (on page 196), showing that our NVMe driver outperforms RDMA.
- A complete performance evaluation of SmartIO in its entirety is presented in Section 7 of Paper V (on page 169). Here, all parts of the implementation is evaluated from multiple angles, including exhaustive tests of the three sharing methods. Through exhaustive comparison testing, we prove that our SmartIO sharing methods **do not add any performance overhead** compared to using local resources. In addition, we show that our SmartIO framework is not limited to a specific Linux version or any specific device, but works for a wide variety of different software versions and devices, by including tests for different Linux distributions, using several benchmarking tools and different kinds of workloads (both synthetic and realistic), and by using different types of PCIe devices.

In this section, we present shortened versions of three selected performance tests from Paper V. The first two tests, presented in Sections 2.4.1 and 2.4.2 respectively, are comparison tests. They prove that our SmartIO framework is able to facilitate the use of remote PCIe devices with the same performance as if these devices were locally installed, by comparing the performance of an I/O-heavy workload running on a local system using a local device, to the same workload running on a remote machine using the same device borrowed through Device Lending. The third test, presented in Section 2.4.3, demonstrates memory optimization capabilities of our proof-of-concept NVMe driver. This is an exploratory test, highlighting device driver functionality enabled by the SISCI API extension.

All three tests use two Intel Xeon machines, and PXH830 NTB adapter cards from Dolphin [11]. Additional PCIe network topologies and hardware configurations are evaluated in the published papers, including tests with larger clusters. In order to create a PCIe topology that is similar for local and remote test runs, we have used a BP-457-ATX PCIe expansion chassis from One Stop Systems. By using an expansion chassis, there are the same number of PCIe switch chips (or "hops") in the path between the CPU and the device in both the local and remote scenarios.[4] Moreover, the switch chips are the same Broadcom PEX8733 [6] chips used in the implementation of the Dolphin NTB adapter cards. Standard and unmodified benchmarking software are used for all three tests.

2.4.1 Device Lending: latency comparison

Using the Device Lending sharing method, machines may use remote resources in the same way they would use local resources. Consequently, it is possible to run a workload locally first, establishing a "baseline" for expected performance measurements. The same workload can then be repeated on a remote system using borrowed devices, allowing us to compare performance measurements to this "local baseline". With all conditions being the same, from the software to the devices being used, any difference in the measured performance for the local run and remote run will reveal whether or not Device Lending adds any performance overhead compared to local access.

Figure 2.12 shows the hardware configurations for the two test scenarios:

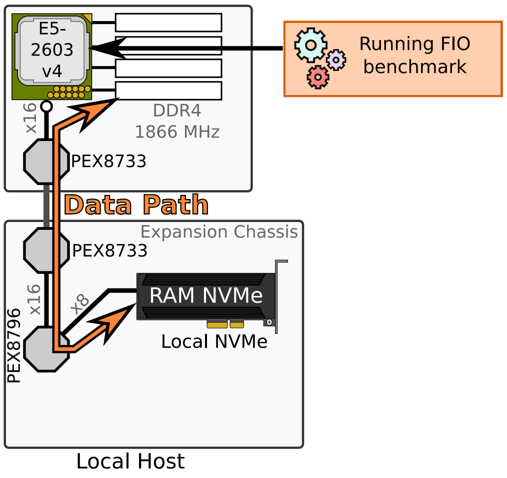
Local Baseline (shown in Figure 2.12a): An external expansion chassis with the NVMe installed, connected to a local machine. The expansion chassis is connected upstream using One Stop System's HIB68-x16 target adapter cards and external PCIe cables. These adapters use the same Broadcom PEX8733 PCIe switch chip used in the Dolphin PXH830 NTB adapters. The IOMMU is disabled, in order to make the configuration comparable to the Device Lending scenario described below.

Device Lending (shown in Figure 2.12b): Two machines connected together in a back-to-back topology, using Dolphin PXH830 adapter cards and external PCIe cables. The remote NVMe is borrowed using Device Lending. The IOMMU is disabled on both the lender and borrower. The same expansion chassis configuration as in the local baseline scenario is used, and since the lender's IOMMU is disabled, PCIe transactions are routed peer-to-peer as illustrated in the figure.

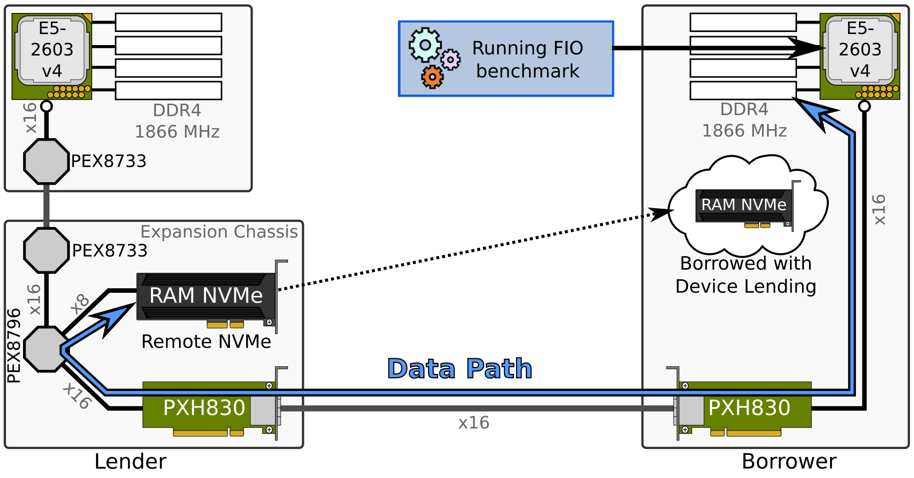
We used the Flexible I/O tester (FIO) [3] to create a synthetic storage workload and measure the latency of reading from disk. FIO is a widely used user space application for benchmarking the performance of storage devices, such as NVMes. In both scenarios, the machines run CentOS 7 with a 3.10 kernel, and version 3.7 of FIO (as available from the CentOS 7 software repositories). Additionally, the standard in-kernel NVMe driver is used in both scenarios.

We configured FIO to perform 655360 reads, where each read is a page-sized block (4 kB) at a random offset on disk, also known as a "random read" pattern.

[4]We show how longer PCIe paths affects DMA performance in Section 7.2 of Paper IV (on page 123) and Sections 7.1.3 and 7.1.5 of Paper V (on pages 175 and 179).



(a) **Local Baseline**: an NVMe in an expansion chassis attached to the local PCIe bus.



(b) **Device Lending**: a borrowed NVMe, appearing local to the remote system.

Figure 2.12: Hardware configuration for the two scenarios in our latency comparison experiment. By using an expansion chassis, the NVMe is the same number of "hops" away from the CPU using the device for both the Local Baseline and Device Lending scenarios. The only difference is whether the switch chips are configured in transparent mode or NTB mode. The data path is illustrated for both scenarios.

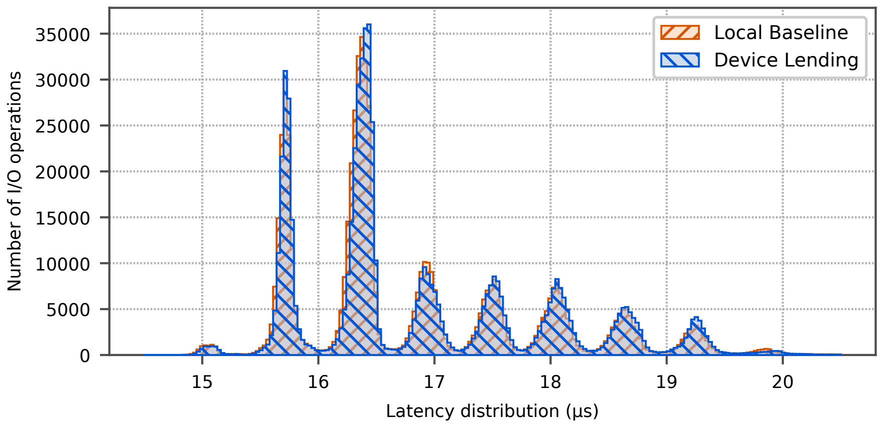


Figure 2.13: Histogram of the latency distributions for reads from storage for both the Local Baseline and Device Lending scenarios. The distribution for both scenarios overlap, demonstrating that our implementation does not add any overhead in the critical I/O path.

Additionally, as the purpose of the test is *not* to benchmark the NVMe itself, but rather any potential overhead of our Device Lending sharing method, the disk used in our experiment is a prototype RAM disk with an NVMe controller from Microsemi. We used a RAM disk to avoid any effects caused by prefetching and caching that modern solid-state flash memory storage devices (SSDs) are capable of.

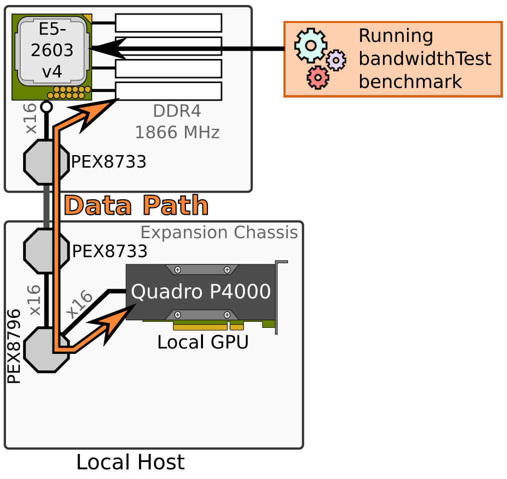
Figure 2.13 shows the latency distribution of read operations for both a local NVMe (Local Baseline) and when accessing it remotely using Device Lending. Each data point is the latency for a full 4 kB read operation. We see that the two distributions overlap, proving that there is no difference in performance for local and remote. A more in-depth explanation of this experiment can be found in Section 7.1.1 of Paper V (on page 171).

2.4.2 Device Lending: throughput comparison

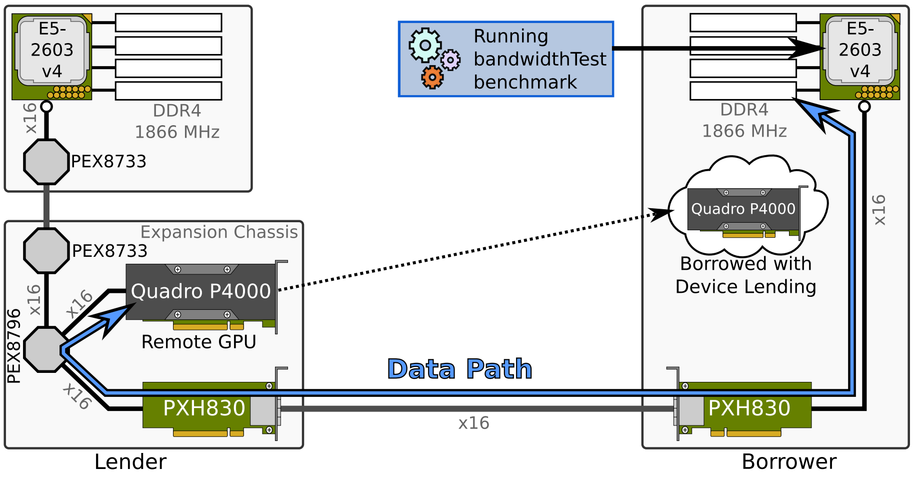
Another performance metric is throughput, particularly the throughput for transferring data between a device and RAM. By performing large DMA transfers, the PCIe links are saturated with transactions and we also stress system memory. This would reveal any overhead caused by our Device Lending implementation that is only visible under high load.

Figure 2.14 illustrates the hardware configuration used in our throughput test scenarios:

Local Baseline (shown in Figure 2.14a): A local machine using a local Nvidia Quadro P4000 GPU (installed in an expansion chassis). As with the latency



(a) **Local Baseline**: using a local GPU in an expansion chassis.



(b) **Device Lending**: using a borrowed GPU.

Figure 2.14: Hardware configuration for the two scenarios in our DMA throughput comparison experiment. The expansion chassis adds the same number of "hops" between the CPU and the GPU in both scenarios.

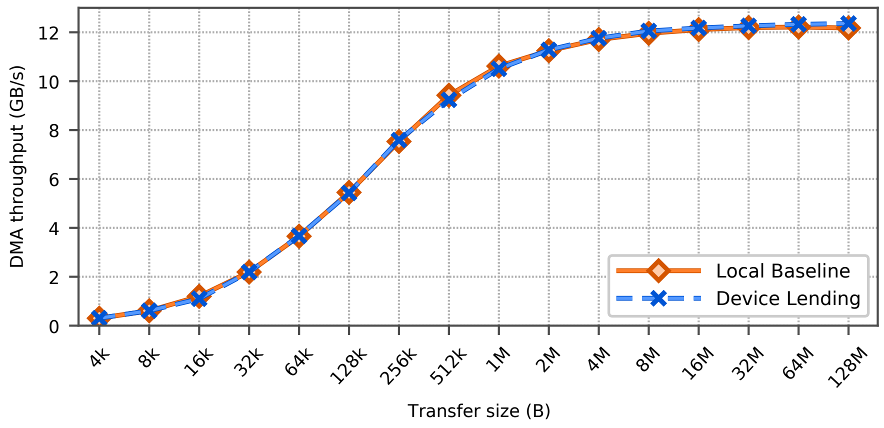


Figure 2.15: The median DMA write throughput of different transfer sizes for the Local Baseline and Device Lending scenarios. The measured performance is the same for both scenarios, demonstrating that our implementation does not add any overhead.

test, we use an expansion chassis to make the PCIe path similar to the Device Lending scenario described below. The IOMMU is enabled.

Device Lending (shown in Figure 2.14b): Two machines connected back-to-back using Dolphin PXH830 NTB adapter cards. One machine is borrowing the Quadro P4000 GPU. The IOMMU on the lender is disabled, in order to allow peer-to-peer DMA transfers as depicted in the figure, while the borrower's IOMMU is enabled.

Both machines are running Ubuntu 18.04.02 with the 4.15 version of the kernel, and we used CUDA 10.1 (with the 418.39 version of the Nvidia driver). In order to create a workload for the GPU, we used the *bandwidthTest* program included in the CUDA Toolkit sample programs [38]. This CUDA program uses the GPU's on-board DMA engine to copy data from the GPU to system memory. For both scenarios, we configured bandwidthTest to initiate 1000 DMA writes to RAM, and repeated this for sizes from 4 kB up to 128 MB in order to reveal any trends that emerge when the transfer size increases.

Figure 2.15 depicts the results of our test. The transfer sizes are plotted along the X-axis, and for each transfer size we plotted the median of the 1000 transfers. The achieved throughput for DMA writes to system memory is almost identical for the Local Baseline and Device Lending scenarios, demonstrating that our Device Lending implementation does not introduce any overhead in the performance-critical path. This experiment is described in further detail in Section 7.1.2 of Paper V (on page 173).

2.4.3 Proof-of-concept NVMe driver experiment

Both the Device Lending and MDEV sharing methods make it possible for remote devices anywhere in the cluster network to be operated by native device drivers. Since most device drivers are written in a way that assumes exclusive control of a device, devices shared with these methods may only be used by a single borrower at the time. For distributed cluster applications, this poses a challenge. For example, if a workload needs to access data on a storage device or to perform computations on a GPU, the application process must interact with the device driver. If the driver is running on a different machine, the application must synchronize with this machine in some way. To further complicate matters, DMA buffers are allocated at the discretion of a device driver, and how these buffers are used is outside our control. Potentially, a device driver may copy data from application memory to internal DMA buffers before it initiates DMA transfers. This can result in poor performance, particularly in the case where the data resides in a memory segment and may be copied between different memory buffers in different machines.

It is possible to remedy this problem by using our extension to the SISCI API to develop a device driver as part of the distributed application itself. Although developing a new device driver may not necessarily be a viable approach in all circumstances, enabling devices to use native DMA to (remote) memory segments may provide performance benefits for cluster applications. To demonstrate this, we have designed the following experiment for our proof-of-concept NVMe driver (as illustrated in Figure 2.16):

- We developed a user space CUDA application, integrating our proof-of-concept NVMe driver as part of this application. The application process is running on the borrower machine, using both the local Nvidia P600 GPU and the remote Nvidia P4000 GPU, as well as a remote Intel Optane P4800X DC NVMe. The remote GPU is borrowed using Device Lending, and both GPUs are managed by the native CUDA drivers (CUDA version 10.2). The NVMe is operated by our application process.

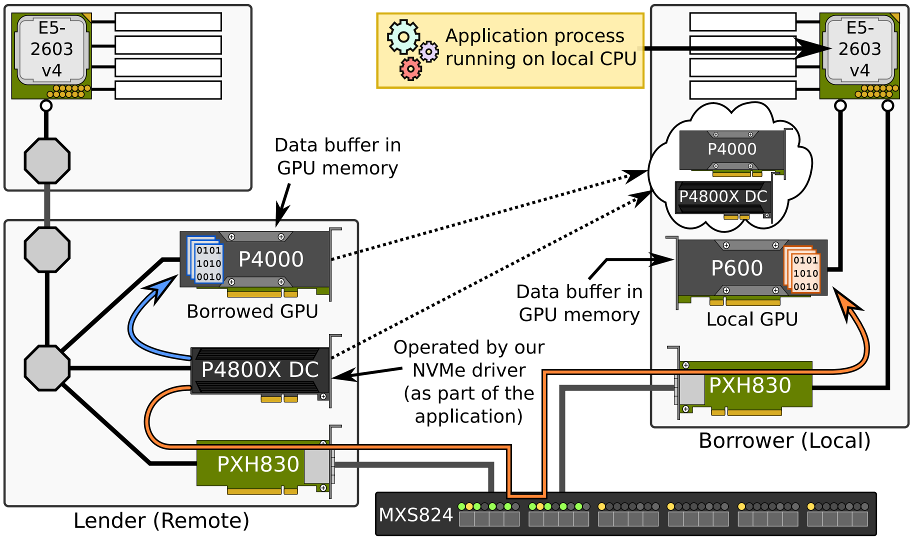
- The purpose of the experiment is to measure how moving the SQ closer to the NVMe affects the latency of I/O operations (as depicted in Figure 2.16b). As such, we allocate a single CQ in local segment in the borrower's RAM. Depending on the scenario, a corresponding SQ is also allocated in one of the following memory locations:

  Borrower RAM: the SQ is allocated in local memory segment on the same machine where our application process is running.

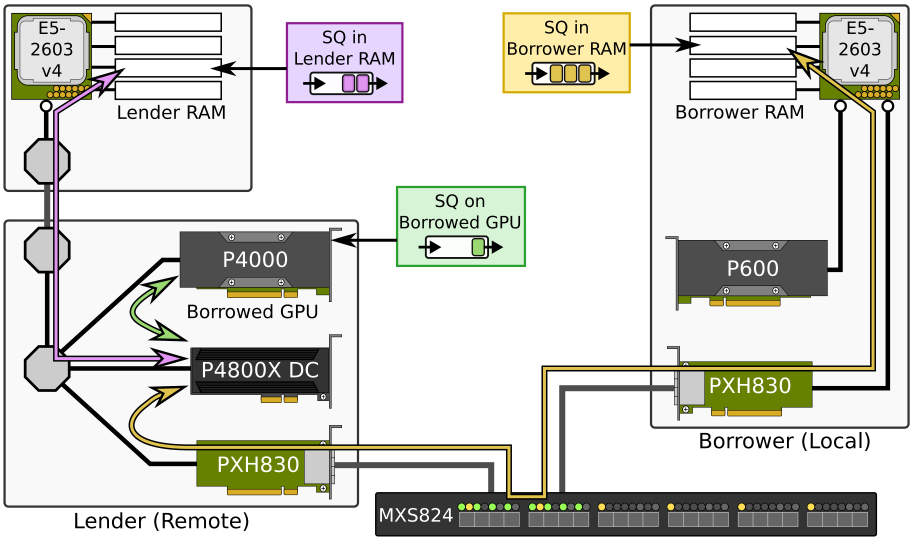
  Lender RAM: the SQ is allocated in lender's RAM, by allowing SmartIO to choose based on access pattern hinting.

  Borrowed GPU: the SQ is allocated in memory on the borrowed GPU.

  For more context on why DMA reads are affected by longer distances, please refer to Section 7.1.3 of Paper V (on page 175).


E5-
2603
v4
Application process running on local CPU
E5-
2603
v4
Data buffer in GPU memory
P4000
P4800X DC
P4000
Borrowed GPU
P600
Data buffer in GPU memory
Local GPU
P4800X DC
Operated by our NVMe driver (as part of the application)
PXH830
Borrower (Local)
PXH830
Lender (Remote)
MXS824

(a) Our proof-of-concept NVMe driver is implemented as part of the application process, running on the borrower's CPU and operating the remote NVMe. The remote GPU is borrowed using Device Lending, and both GPUs are operated using the native CUDA driver (also running on the borrower). Data is loaded from storage directly into buffers in GPU memory.



(b) We measure the latency effect of moving the SQ closer to the NVMe. This reduces the distance the NVMe needs to read across in order to fetch I/O commands.

Figure 2.16: The design of our proof-of-concept NVMe driver experiment.

- As illustrated in Figure 2.16a, the application process allocates a memory buffer on both GPUs, which are exported as shared memory segments by our SISCI API extension and mapped for the NVMe. Using the SQ, the application loads data from storage on the NVMe directly into the memory buffers on the GPUs by issuing read commands. Since the purpose is to measure the latency of individual I/O operations, data is loaded into GPU memory using a random read access pattern and a block size of 4 kB. First, we read data into memory of the borrowed GPU. Then this process is repeated for the local GPU. For each GPU, we performed 327,680 reads (655,360 in total on both). Note that since we are performing read operations, i.e., loading data from storage, the data destination does not affect latency. However, both GPUs are included as destinations in our experiment, for the purpose of showcasing the capabilities of SmartIO.[5]

- The *command completion latency* for each read operation is measured, and the SQ is configured to have a queue depth of just a single entry, in order to avoid aggregated measurements. We define the command completion latency as the time elapsed from writing a command to SQ memory (followed by a write to the SQ doorbell register) until the corresponding completion entry shows up in CQ memory. Note that the timer is started *before* writing the command to (remote) memory, so the recorded latency value also reflects the delay of writing to a remote memory location.

Both the lender and the borrower are running Ubuntu 18.04.4 with the 4.15 version of the Linux kernel. The IOMMU is enabled on the borrower, while it is disabled on the lender.

Figure 2.17 depicts the distributions of recorded latency values for I/O read operations, for all three scenarios. The same datasets are show as a histogram and as a boxplot. Note that we have adjusted the Y-axis, so suspected outlier measurements are not shown. The median for all three scenarios are marked with horizontal lines. Our results demonstrate that moving SQ memory closer to the NVMe reduces the latency, as the distance the NVMe has read across shrinks (see Figure 2.16b for reference):

Borrower RAM: The NVMe has to read across the internal switch chip in the expansion chassis, the lender's NTB adapter, the cluster switch, and the borrower's NTB adapter (4 "hops"). The measurements for this scenario has the highest command completion latency values.

Lender RAM: The NVMe reads across the internal switch chip, and both the downstream and upstream transparent switches, in order to fetch commands (3 "hops"). It is interesting to note that even though the borrower has to write the longest distance in order to submit I/O commands (6 "hops"), this scenario still has a performance benefit because the distance for the NVMe is reduced.

[5]A similar experiment is also presented in Section 7.4.1 of Paper V (on page 192), where only the local GPU is used as a data destination.

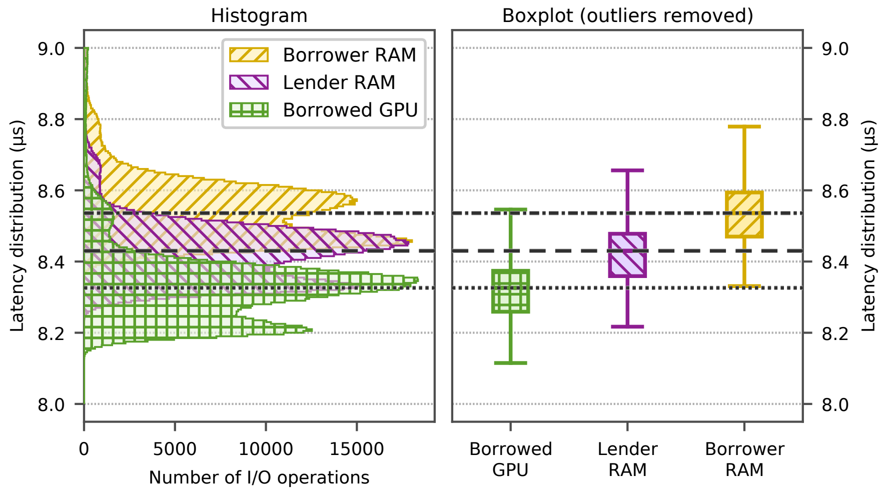


Figure 2.17: Distribution of recorded command completion latencies as a histogram (left) and as a boxplot (right). The closer the SQ is to the NVMe, the lower the latency is.

Borrowed GPU: By using the GPU installed alongside the NVMe in the same expansion chassis, the NVMe only has to read through the internal switch chip in order to fetch commands (1 "hop"). Although GPU memory has different latency characteristics than RAM, our measurements show that this scenario has the lowest command completion latency values.

This experiment shows that there may be performance advantage of building device driver functionality into the application, compared to using Device Lending and native device drivers. While there is an associated development cost of implementing new device drivers using the SmartIO API extension, it becomes possible to control memory locations using access pattern hinting and optimize data flow through the cluster network, for example by streaming data directly into GPU memory as shown in this experiment. Additional functionality also becomes available through the use of the SISCI API, such as relying on PCIe multicasting to replicate data across several cluster machines in a single operation. Replicating NVMe data across 60 machines using multicasting is demonstrated in Section 7.4.2 of Paper V (on page 194).

Our NVMe driver experiment also shows that the different sharing methods of SmartIO can be *combined*. The Nvidia P4000 GPU is borrowed using Device Lending (and operated by the native CUDA driver), but using our API extension its memory is disaggregated, allowing the NVMe to access buffers allocated in GPU memory directly. We argue that this demonstrate the strength of our SmartIO framework, as we are able to combine the traditional I/O model with distributed, shared-memory functionality.

2.5 Related work

Sharing resources efficiently among networked machines is a wide-ranging topic that span several research areas. In fact, each individual component of our solution could potentially be discussed at great length on their own, in order to place them in proper context. However, at its core, SmartIO is a framework for machines in a PCIe network to share their internal devices and memory resources. In this section, we attempt to give a condensed overview of related work we consider the most relevant. A more detailed presentation of related work can be found in Section 9 of Paper V (on page 206). Background for the ideas behind our proof-of-concept NVMe driver is summarized in Section 9.4 of Paper V (on page 210).

2.5.1 Solutions not using NTBs

State of the art within disaggregation and resource sharing can broadly be divided into three categories:

- So-called “rack-scale disaggregation” by **logical partitioning**, where modular blade servers are connected to a backplane or a shared I/O bus fabric within a data center rack. Devices are installed in dedicated resource servers, and can be dynamically assigned to compute hosts as needed. These solutions can be realized using interconnection technologies that are specialized for this purpose [24, 25, 49, 51], but PCIe (with additional virtualization support) is the most used alternative for commercial solutions [7, 17, 31, 44]. We also include Multi-Root I/O Virtualization (MR-IOV) [39] in this category, even though we are unaware of any available implementations of it.

  PCIe-based solutions use switch chips with virtualization support that makes it possible to isolate devices and CPUs by creating logical device trees for each host [7, 35, 64, 65], as illustrated in Figure 1.2. Some solutions even support assigning individual VFs of an SR-IOV-capable device to different hosts [7]. However, as we addressed in Section 1.1, it is only possible to distribute resources that are directly attached to these partitionable switch chips. Sharing the *internal* resources of a machine, such as its memory and local devices, is not possible. Consequently, solutions based on switch partitioning cannot support memory-to-memory communication between hosts, nor are they able to support any memory disaggregation. Instead, additional disaggregation solutions like RDMA are needed.

- Memory disaggregation through **remote memory access**. The goal of most memory disaggregation solutions is not necessarily to make more resources available or improve resource utilization in the cluster, but rather to facilitate shared-memory communication for distributed cluster applications. Remote memory access is most commonly implemented on top of RDMA, and made available to the application through typical parallel programming models, such as message-passing [23] and remote procedure calls [32], or through more explicit abstractions [2]. Approaches for accessing remote memory in a more transparent fashion also exist, usually achieved by modifying the system page fault handler to initiate RDMA transfers so remote memory pages can be "faulted in" [18, 26, 27].

Especially message-passing protocols implemented on top of RDMA over InfiniBand appear to be widely used for distributed computing. Many of these message-passing solutions also support GPUDirect, effectively making it possible to disaggregate GPU memory [46, 62, 63]. Although RDMA enables efficient data transfer over a network through one-sided initiation and direct access to application memory, it nevertheless introduces a layer of indirection; compared to a CPU (or device) reading and writing to a memory-mapped location directly, latency is significantly increased by needing to initiate (and wait for) RDMA transfers.

- Device disaggregation by **distributed I/O over a network** using RDMA. Examples include rCUDA for sharing GPUs in an InfiniBand cluster [12], and NVMe-oF using RDMA [15, 19]. Solutions for sharing GPUs [20] and Ethernet NICs [61] over a PCIe fabric have also been proposed, by using PCIe switch chips capable of DMA to transfer data from memory to memory in the different machines.

  As mentioned in Section 1.1, RDMA-based approaches must communicate with a device driver running on the device-side system, in order to interact with the remote device. As this is typically solved by implementing a middleware service that uses an existing driver or by implementing a distributed driver, these solutions are generally specific to the (type of) device they disaggregate. Moreover, as illustrated in Figure 1.4, this additional software component on the remote system inevitably leads to a performance overhead, compared to a local device driver directly interacting with a device.

With its three sharing methods, our own SmartIO framework intersects all three of these disaggregation categories. Similarly to PCIe-based rack-scale solutions, we are able to distribute devices using the Device Lending and MDEV sharing methods (Sections 2.3.3 and 2.3.4), including individual VFs of devices capable of SR-IOV. We also have an inherent relationship with memory disaggregation solutions, through the shared-memory functionality provided by the SISCI API and our SmartIO extension of it (Section 2.3.5). It is even possible to disaggregate devices similar to RDMA-based device disaggregation, as demonstrated by our proof-of-concept NVMe driver (Section 2.3.6). However, by being implemented on top of PCIe cluster networking, SmartIO is able to offer significant improvements over the other three types of disaggregation:

- Unlike PCIe-based rack-scale disaggregation, where only resources attached directly to partitionable switch chips can be shared, SmartIO instead makes it possible for all machines to lend out their internal devices and borrow resources from remote machines.

- In contrast to RDMA-based solutions for memory disaggregation, remote memory segments can be mapped into a local software application's virtual address space directly using the SISCI API. Using our extension to the SISCI API, we greatly extend this functionality by making it possible to map shared memory segments for *devices* as well. The need to use RDMA for shared-memory communication is removed entirely, as both CPUs and devices may instead read and write to remote memory directly. Moreover, it becomes possible to integrate I/O and device operation into the cluster application itself.

- Contrary to device disaggregation using RDMA, SmartIO removes the need to interact with a remote device driver on the device-side system. Instead, as SmartIO enables access to remote resources over native PCIe for both driver and device, devices can be operated directly by a device driver running locally on the borrower. As no software is needed in the performance-critical data path, SmartIO has the performance advantage of native PCIe.

- Using either the Device Lending or MDEV sharing methods, devices appear locally installed. This makes SmartIO a more general solution compared to RDMA-based disaggregation, in the sense that *any* PCIe device may be shared and operated by existing device drivers. However, in addition to distributing devices to both physical machines and VMs, we also facilitate what we call "MR-IOV in software" through the SISCI API extension. This allows non-SR-IOV devices to be shared with several hosts at the same time, something that is not possible with current solutions based on partitionable PCIe switch chips and requires RDMA. SmartIO is "the best of both worlds", by combining the flexibility of shared-memory functionality with the ability to use remote devices as if they were locally installed.

- Disaggregation of device memory is made simpler by SmartIO. Because SmartIO abstracts away the physical PCIe topology, the device memory of both local and borrowed devices is trivially memory mapped by an application and for other devices using our SISCI API extension. For example, applications that rely on GPUDirect can memory map (remote) GPU memory directly, instead of needing to use abstractions such as message-passing. Moreover, since it is also possible to borrow remote GPUs using Device Lending, SmartIO supports CUDA's native unified memory model [47] as well, allowing remote GPUs to read and write to each other's memory as if they were installed in the same machine. This makes application development significantly easier, as software can be written as if all resources are local. We are not aware of any RDMA-based GPU disaggregation solutions that are able to support this.

In short, our SmartIO framework provides several advantages over existing disaggregation solutions. Disaggregation solutions based on RDMA introduce additional software complexity and indirections that lead to a disparity in performance, compared to using resources over native PCIe. PCIe-based disaggregation solutions do not have such performance issues, but are limited to sharing devices installed in dedicated servers, as they lack the shared-memory capabilities necessary for sharing the *inner* resources of individual machines. We present a more technically detailed comparison between specific solutions and our own SmartIO implementation in the published papers, particularly in Section 9 of Paper V (on page 206).

2.5.2 Solutions using NTBs

Solutions that enable disaggregation using NTBs are particularly interesting to us, due to the similarities with our own work. By using the same Broadcom 8733 [6] PCIe switch chips used in the Dolphin PXH830 adapter cards used in our own evaluation (Section 2.4), Shim et al. [48] and Lim et al. [28] have implemented NTB host adapter cards and connected three machines in a cluster. They have extended the OpenSHMEM API with support for their NTB implementation. However, their focus seem to be enabling partitioned global address space-style shared memory functionality for high-performance computing applications, and distributing and sharing devices appears not to have been considered as part of their solution. It should also be noted that the underlying memory-mapping functionality provided by their solution is very similar to functionality already existing in the SISCI API.

The Ladon system [60] facilitates access to the same SR-IOV device from multiple VM guests. Several machines and a SR-IOV device is connected to a top-of-rack PCIe switch with NTB-capable ports. The device and a dedicated "management host" is connected to the switch in *transparent* mode, so that the management host may enumerate the PCIe bus and configures the device. Multiple "compute hosts" are connected to the same switch through *non-transparent* switch ports, i.e., NTBs. The management host maps the entire memory of each compute host for the device, and assists the hypervisor on each compute hosts in mapping device BARs, in order to pass-through individual VFs to VMs running on the compute hosts. It is also possible to map interrupts directly into the VM guests [57, 59].

The Ladon system is very similar to our own MDEV implementation. For example, the management host is comparable to our lender, and the compute hosts would be the equivalent to host machines that run VM borrowers in SmartIO. However, Ladon and our own SmartIO framework differ in some areas:

- Only VM pass-through appears to be have been considered for the Ladon system. SmartIO, on the other hand, supports sharing with both physical host machines and VM guests, through the Device Lending and MDEV sharing methods respectively. Because the management host maps the *entire* RAM of each compute host, it could in theory be possible to extend Ladon with support for physical machines. However, such an extension would likely require that device drivers are adapted to interact with the management host to set up mappings. In contrast, our own Device Lending sharing method makes it possible for bare-metal hosts to use devices without requiring modifications to existing device driver software.

- Perhaps the main difference between Ladon and SmartIO is that while a single host, i.e., the management host, owns the device in Ladon, our SmartIO system is truly distributed by supporting multiple machines acting as lenders. Machines may even act as both lender and borrower at the same time. Moreover, in Ladon, the management host becomes a single point of failure. Ladon has since been extended with fail-over support, allowing a back-up management host to replicate the PCIe fabric enumeration of the first host, and seamlessly take ownership of the device in case the first management host fails [58]. However, we argue that this still does not make Ladon distributed in the same sense as SmartIO. For example, it is not possible for a compute host to use devices from *different* management hosts. In contrast, SmartIO supports scaling out and using devices from several machines across an entire cluster.

- The Ladon implementation relies on mapping the entire memory of all compute hosts for the device. Because of this, the number of compute hosts that can be supported in the Ladon setup will be limited to a handful of hosts due to the combined BAR size requirements of the NTBs. This differs from our own MDEV implementation, where only the VM memory is mapped for the device. Additionally, should the NTB's BAR size become a limitation, it is possible to add more lender machines and borrow devices from multiple lenders.

Compared to other solutions implemented using NTBs, SmartIO is a more comprehensive framework for sharing devices and memory resources. Our solution makes it possible to share devices to remote machines (both bare-metal host machines and VMs), as well as and disaggregating memory resources. Using SmartIO, it is even possible to combine sharing capabilities with shared-memory functionality, allowing devices to become part of the same global address space as distributed, shared-memory applications.

Chapter 3

Conclusion

As distributed and parallel computing applications are becoming increasingly compute-intensive and data-driven, I/O performance demands are ever growing. Computing accelerators (such as FPGAs and GPUs), high-throughput NICs, and fast storage devices like NVMes, are now commonplace in most modern computer systems. Nevertheless, distributing such I/O resources in a way that maximizes both performance and resource utilization is a challenge for heterogeneous computing clusters. To avoid that individual machines becoming performance bottlenecks, resources must be shared efficiently between machines in the cluster.

In this dissertation, we have addressed this challenge and presented our SmartIO framework for sharing I/O resources between machines connected over PCIe. Our SmartIO framework effectively makes all machines, including their internal devices and memory, part of a common PCIe domain. Resources in remote machines can be used as if they were installed locally, without any performance degradation compared to local access, and without requiring adaptions to device drivers or application software. The hard separation between local and remote is blurred out, as machines can freely share their internal devices and memory resources with other machines in the cluster.

3.1 Summary

Connecting two or more computer systems over PCIe is possible by using PCIe NTBs. NTBs have memory address translation capabilities that makes it possible for a machine to map segments of remote memory directly into local address space. However, leveraging NTBs to share the internal devices and memory of a machine with other, remote machines is a challenge, as the use of a remote resource requires software to be aware of the fact that the resource is on the other side of an NTB. For example, a device driver operating a remote device must use addresses that correspond to the remote device's address space when initiating DMA transfers or configuring interrupts. This additional complexity makes it infeasible to rely on NTBs alone to implement a resource sharing solution, as it would require extensive modifications to existing software.

To solve this, we have developed our SmartIO framework for sharing devices and memory resources between machines connected with NTBs. Our solution consists of "lenders", machines lending out one or more of its internal devices, and "borrowers", machines using such a device. Machines can act as lender and borrower at the same time, making SmartIO fully distributed. Any type of PCIe device may be shared, as SmartIO is built on standard PCIe. SmartIO keeps track of which machines devices and memory segments reside in, and is able to map resources on behalf of devices and resolve memory addresses as they are seen by devices. As such, SmartIO provides a logical decoupling of devices and which lender machines they are installed in, solving the challenge of managing multiple address spaces and making remote resources appear and behave as if they are local.

SmartIO supports three different methods of device sharing:

- Our **Device Lending** sharing method makes it possible to dynamically assign a PCIe device to a remote borrower machine. The fact that the device is remote is made transparent to the system, allowing the device to be used by native device drivers and application software as if it was locally installed.
- Our **MDEV** extension to the KVM hypervisor makes it possible to distribute devices to VMs running on remote machines, by facilitating *pass-through* of a device to the VM guest. Application software and device drivers running inside the VM guest can directly interact the physical device, without compromising the isolation of the virtualized environment.
- Our **SISCI API extension** makes it possible to disaggregate devices and memory resources in software. Using this API extension, we have also implemented a **proof-of-concept NVMe driver** that demonstrates sharing NVMes with multiple machines at the same time.

We have performed an extensive performance evaluation, consisting of a comprehensive collection of synthetic performance benchmarking and realistic workloads. We have made a point out of using standard benchmarking software and device drivers, as well as a wide variety of PCIe devices, in order to demonstrate the completeness of our SmartIO framework. Particularly, we have performed comparison tests where we compare the performance of a workload using remote resources to the same workload running only on a local system. The results prove that, when conditions are similar, the SmartIO sharing methods **do not add *any* performance overhead** compared to using local resources. Furthermore, we have also explored how different network topologies affect the performance, and have identified situations where the IOMMU can become a potential performance bottleneck. Finally, our exhaustive performance test suite also includes tests using our proof-of-concept NVMe driver that highlights possibilities that are enabled by our shared-memory approach to device sharing.

3.2 Revisiting the problem statement

The main goal of this dissertation was to use NTBs to develop a solution that allows the internal I/O resources of machines to be shared with, and used by, remote machines in a cluster, as if these resources were local to the remote machines using them. In Section 1.2, we broke down the challenges of this goal into six objectives:

Objective 1: Ubiquitous sharing in the cluster should be supported, allowing any machine to contribute any of its internal PCIe devices, and allowing any machine to be able to use shared devices, even contributing and using devices at the same time.

SmartIO is fully distributed, allowing *any* machine in the cluster to act as a "*lender*" or a "*borrower*", or even acting as both at the same time. Any PCIe device may be registered with SmartIO and shared, as demonstrated by our comprehensive performance evaluation in Paper V. As such, we enable a peer-to-peer sharing model, where all machines in the cluster can participate in the sharing through contributing their own resources and using resources shared by others.

We implemented three different sharing methods for our solution:

- The Device Lending sharing method, explained in Section 2.3.3, makes it possible to distribute devices to remote machines. The initial Device Lending method is presented in Paper I. Subsequent improvements are presented in Papers III to V.
- The MDEV extension to the KVM hypervisor makes it possible to distribute devices to VMs running on remote machines, as detailed in Section 2.3.4. The initial MDEV method is presented in Paper III, and improved versions are presented in Papers IV and V.
- The API extension brings device-oriented programming semantics and device driver support functions to the SISCI API. Using the API extension, user space device drivers can be implemented using the same API used to implement shared-memory communication using NTBs, as explained in Section 2.3.5. The API extension is presented in Paper V.

These sharing capabilities set SmartIO apart from existing PCIe-based disaggregation solutions (including Ladon [60]), as these solutions are only able to share devices in dedicated servers. Thus, our sharing methods solves Objective 1.

Objective 2: The fact that resources may be remote should be functionally transparent, allowing systems to use remote resources in the same way as if they were local, without requiring any modifications to device hardware, device drivers, host OS, or application software.

Our three sharing methods address this objective in the following ways:

- Device Lending inserts a remote device into the local device tree of the host OS by using a "shadow device". This allows device drivers, application software, and even the (host) OS itself to use the remote device through *native* OS interfaces, in the same way they would use a local device. No adaptations to existing software is required. This is further explained Device Lending in Papers I and III to V.

- MDEV enables pass-through of a remote device to a VM. Software running in the guest, including device drivers and the guest OS, may interact with the physical device directly, as if the device was locally installed. No modifications to VM emulator software or host OS is necessary. MDEV is described in further detail in Papers III to V.
- Using the SISCI API, remote memory segments are mapped directly into the virtual address space of a local application. Our extension to the SISCI API makes it possible to map such segments for *devices* as well. This enables native DMA to remote memory resources, as if both the device and the memory being accessed were both installed in the same, local machine. Moreover, using the API extension, the physical location of both devices and memory segments are abstracted away. User space device drivers implemented using our extension can be written as if all resources are local, similarly to how a local user space device driver (for a local device) would be implemented. The API extension is described in Paper V.

Whether resources are remote or local is made transparent by SmartIO, as remote devices and memory resources both appear and behave as if they are locally installed. In this regard, SmartIO differs from existing disaggregation solutions based on RDMA. Contrary to these solutions, we do not require interacting with a device driver running on the remote system, thus avoiding any middleware services or specialized adaptations to existing software. Scaling out becomes significantly easier, as SmartIO allows remote resources to be used natively instead. Thus, this aspect of SmartIO solves Objective 2.

Objective 3: The fact that resources may be remote should be transparent with regard to performance, remote resources should be used with native PCIe performance, and as close to local access as possible.

One of the main challenges for our Device Lending and MDEV sharing methods was that local RAM must be mapped ahead of time in order to avoid communication overhead in the performance-critical path, yet memory used by a device driver can not be known in advance. To overcome this, our SmartIO implementation supports using the borrower's IOMMU to create continuous memory ranges that can be mapped as “DMA windows” through the lender's NTBs before use. Memory pages can then be dynamically added and removed from these IOMMU ranges locally on the borrower, and communication with a remote system in the critical path is avoided.

Once mapped, remote resources are accessed with native PCIe performance, as all address translations are done in NTB (and IOMMU) hardware. In fact, our evaluation in Paper V prove that, when conditions are similar, SmartIO allows remote resources to be used *without any performance overhead* compared to using local resources. Nevertheless, using remote resources may lead to a longer distance between resources. As such, there are some caveats that must be considered:

- Longer PCIe paths affect DMA performance, particularly DMA reads, as we uncovered in Papers III to V. This remains an unsolved challenge for Device Lending or MDEV, as we have no control over the memory allocated by a device driver in these instances. Therefore, we recommend considering the length of PCIe paths when designing the cluster. The issue of longer PCIe paths affects drivers implemented using our SISCI API extension to a lesser extent; by using memory access pattern hinting when allocating DMA buffers, SmartIO will attempt to minimize the distance a device or a CPU needs to read across. The performance experiment presented in Section 2.4.3 demonstrates this.

- Our performance experiments in Papers III to V also revealed that an IOMMU in the data path can negatively affect DMA performance, as the IOMMU may split large PCIe transactions into several, smaller-sized transactions. This is especially an issue for our MDEV sharing method, as SmartIO uses the lender's IOMMU in order to map the device to the same guest-physical address space as the VM the device is passed-through to. For Device Lending, the use of an IOMMU on the lender is optional. However, the use of an IOMMU on the borrower is necessary (except in a few scenarios where it is possible to map the entire RAM of the borrower). Consequently, this may introduce limitations on scenarios where machines act as both lenders and borrowers, where maximizing DMA performance is a requirement. In the case where device drivers are implemented using the API extension, an IOMMU is entirely optional on *both* the lender and the borrower.

By making it possible for remote resources to be accessed over native PCIe, Objective 3 is solved. Improving performance issues involving IOMMUs is a candidate for future work.

Objective 4: Shared resources should be distributed *dynamically*, and direct access to device memory and system memory should be configured at run-time, also between multiple devices residing in different hosts.

Using SmartIO, resources may be shared without requiring machines to be rebooted. Devices registered with SmartIO can be borrowed by any machine, at any time, using any of the three sharing methods. For example, a machine may borrow a device using Device Lending and at the same time run a VM that is borrowing another device using MDEV. The different sharing methods can also be combined, as demonstrated by the proof-of-concept NVMe driver experiment presented in Section 2.4.3. When the device is no longer needed, it can be returned so it may be used by another borrower. Through borrowing and returning devices, systems may dynamically scale I/O resources up or down based on current workload demands.

Devices are logically decoupled from the machines they are physically installed in, allowing software to be moved to any machine in the cluster. SmartIO keeps track of both memory segments and devices, and is able to locate resources in the cluster, without requiring that the user knows anything about the underlying PCIe topology. The shortest path between devices, CPUs, and memory segments is determined automatically, and SmartIO configures NTBs along that path in order to map remote memory resources for CPUs and devices. Moreover, SmartIO also supports borrowing devices from multiple lenders and enabling peer-to-peer DMA transfers between them, as we explain in Papers III to V. Peer-to-peer can be enabled when borrowing devices using Device Lending or MDEV, which is demonstrated in the various peer-to-peer experiments presented in these papers. Peer-to-peer is also supported when using the API extension, which we demonstrate in the proof-of-concept NVMe driver experiment (Section 2.4.3). Our various performance experiments demonstrate that SmartIO is a dynamic and flexible sharing framework, thus solving Objective 4.

Objective 5: Disaggregation of system memory, device memory, and device *functionality* should be supported, and the solution should be able to distribute component parts to different hosts, as well as provide software facilities for resources that do not support disaggregation in hardware.

SmartIO is able to disaggregate multi-function devices, such as devices capable of SR-IOV, and distribute individual device functions to different borrowers. An experiment demonstrating this is presented in Paper V. Devices that do not support SR-IOV may be disaggregated in *software* instead, using our extension to the SISCI API. Using the API extension, a device be borrowed by several machines simultaneously. Our proof-of-concept NVMe driver presented in Paper V demonstrate this, where several borrowers share the same (non-SR-IOV) NVMe. In other words, the API extension enables "MR-IOV in software".

The API extension makes it possible to implement device drivers as part of distributed, shared-memory cluster applications. Any memory segment anywhere in the cluster can be mapped for devices, so they may access them directly, including segments in local RAM on the borrower, segments in RAM on the lender, and even segments in memory of a different cluster machine altogether. Device BARs are also automatically exported by SmartIO as shared memory segments, allowing device memory to be mapped for the application process or even for *other devices* (thus enabling peer-to-peer). As such, SmartIO supports disaggregating device memory. It is even possible to map multicasting segments for a device, allowing a device to stream data to multiple destinations in a single operation. Moreover, SmartIO makes it possible to associate memory segments with a device (rather than a machine in the cluster), allowing the location of memory segments to be abstracted away in a similar fashion to devices. This allows software to be written as if all resources are local, and can run on any machine in the cluster. SmartIO is able to optimize memory locations without requiring that the user is aware of the underlying PCIe network topology. The proof-of-concept NVMe driver experiment presented in Section 2.4.3 demonstrates all of these capabilities, proving that Objective 5 is solved.

Objective 6: To prove real-world deployment capabilities, the solution should be tested on realistic and relevant workloads and benchmarks.

To prove that SmartIO is an efficient solution for real-world applications, we have performed a comprehensive performance evaluation consisting of both synthetic microbencmarks and realistic, large-scale workloads. All parts of our SmartIO implementation have been evaluated, and we have included several sharing scenarios and network topologies have been evaluated, as well as a wide range of standard and commodity PCIe devices like GPUs, NICs, and NVMes. As SmartIO makes it possible to use remote devices as if they were local, we have used standard and unmodified benchmarking tools and device drivers. Through comparison testing, we prove that SmartIO does not add any performance overhead compared to using local resources, in terms of latency and throughput. We have also explored the performance effects of moving resources further away, and present a thourough analysis of this. Finally, a proof-of-concept NVMe driver was developed in order to evaluate our API extension and the disaggregation possibilities enabled by SmartIO. All of the published papers contributed towards solving this objective (Papers I to V).

By solving all of our six research objectives, we have answered the central research question of this dissertation:

> *Can NTBs be leveraged to allow the internal memory and devices of individual computers in a PCIe-networked cluster to be shared with and used by remote machines in the cluster, as if these resources were local to the remote machines?*

We have not just implemented yet another disaggregation solution, but developed a new and more flexible solution by taking a novel approach: utilizing the memory mapping capabilities of NTBs to unify traditional device I/O with distributed, shared-memory computing. Our implementation makes it possible for machines to share their inner devices and memory with other machines in a PCIe cluster. Remote resources can be used over standard PCIe, making our SmartIO framework a zero-overhead solution for scaling out and transparently using more hardware resources than there are available in a single machine. By lending out their own local resources and borrowing remote resources, machines take part in a dynamic and composable I/O infrastructure. Thus, we have shown that we can leverage NTBs to develop a solution where resources can be freely shared in a PCIe-networked cluster.

3.3 Future work

Several ideas for improvements emerged during the development of our SmartIO framework. Here, we highlight some areas that warrant further investigations and outline some possible directions for both ongoing and future work:

- The security implications of allowing remote machines to use and control internal system resources is something that should be addressed. By lending away local devices, the lender effectively yields control over it to software running on a remote system. A flawed device driver may cause a device to read from and write to rogue memory addresses, potentially crashing the system. A malicious driver may even *intentionally* overwrite memory, or misuse DMA in order to snoop data from memory on the lender. This is particularly a concern in the context of our SISCI API extension, as this exposes device functionality to user space software. Using an IOMMU on the lender offers some protection against undesired memory accesses, as devices are isolated in their own virtual address space. However, as IOMMU domains only isolate *per device*, it could still be possible for a malicious program to interfere while others are using a device, and something like Process Address Space ID (PASID) may be required. The security challenges of one-sided initiated I/O is an understudied topic in general [56], so any work in this area would likely be a significant contribution.

- Our performance experiments presented in Papers III to V show that a combination of using the lender-side IOMMU and long PCIe paths may severely impair DMA performance. While the use of an IOMMU on the lender is optional for Device Lending, it is a requirement when using our MDEV sharing method. Additionally, since using an IOMMU on the borrower is required by Device Lending in most scenarios, the IOMMU will be present in machines that are *both* lenders and borrowers. Reducing or eliminating IOMMU performance penalty is a strong candidate for future improvements. Alternative CPU/IOMMU architectures should be investigated, to determine if they behave similarly to the Intel Xeon CPUs used in our experiments. Implementing support for Address Translation Services (ATS) [40] should also be considered, as ATS allows devices (and PCIe switch chips) to cache I/O addresses resolved by an IOMMU. However, it should be mentioned that since ATS requires support in both devices and IOMMUs, it appears to not be widely adopted, especially for commodity hardware.

- Our MDEV implementation currently supports so-called "cold migration". VMs can shut down, migrate, and restart on a different host, while keeping the same passed-through devices. If the VM emulator supports it, it could also be possible to support hot-adding and hot-removing devices to a running VM, making live migration theoretically possible by first removing all devices, migrating, and then re-attaching them afterwards. However, this would temporarily disrupt device I/O and force guest drivers to reset all devices. Supporting real "hot migration", remapping devices while they are in use, with minimal disruption, is something we wish to implement in future work. Not only would such a solution require keeping memory consistent during the migration warm-up, but a solution would also need to consider DMA transactions in-flight during the migration. A mechanism for re-routing transactions, without violating the strict ordering required by PCIe, should be implemented, and would most likely require hardware support that does not exist today.

- While our proof-of-concept NVMe driver demonstrates that it is possible for multiple machines to share the same NVMe simultaneously, it is not very practicable by itself; as our implementation only provides block-level access to user space cluster applications, implementing a file system or coordinating access is currently the responsibility of the application. Therefore, we are currently working on implementing our proof-of-concept prototype as a *kernel space* driver, making the disaggregated NVMe available to the system for general use. This way, the disaggregated NVMe can be used as a system disk, making it possible to format it with existing Linux file systems. Moreover, since the NVMe can be shared with multiple machines, it would also be possible to use a shared-disk file system, such as GFS2. This new kernel space implementation could co-exist with the existing user space implementation, as queues can be assigned to application processes and kernel modules alike.

Finally, future interconnection technologies that are built on PCIe could provide new opportunities, and should be explored once they become widely available. Particularly Compute Express Link (CXL) 2.0 [53] is interesting in the context of memory disaggregation, as it provides new cache-coherent protocols for accessing system and device memory.

Our critical review of the goals and objectives showed that such sharing of resources across cluster machines is possible. However, there also are unsolved questions, and we have presented some additional potential directions. Nevertheless, we believe that our research results are a step in the right direction and should be a sound foundation for further research activities.

Glossary

API extension

One of the sharing methods of SmartIO, extending the SISCI API with device-oriented functionality for writing device drivers as part of shared-memory cluster applications. 15, 17, 23–25, 30–35, 38, 44, 46, 47, 50, 54–60

Base Address Register (BAR)

Registers in a device's configuration space containing the start addresses of the memory regions of a device. The term is often used synonymously for the device memory region the BAR register describes. 19–22, 25–28, 30, 32, 34, 35, 51, 52, 58,
see also configuration space

borrower

A computer system using a remote device with SmartIO. 23–25, 27–31, 34, 36, 39, 43, 44, 46, 50–58, 60,
see also lender

configuration space

A set of standard registers that is used to configure a PCIe device, such as reserving BARs and interrupt addresses. 20, 28, 29

CUDA

The parallel computing platform and API of Nvidia GPUs for general purpose processing. 13, 15, 36, 37, 43–45, 47, 50

PCIe device function

PCIe devices may have one or more functions, each appearing to the system as individual devices with their own set of resources. Also known as a PCIe endpoint. 15, 21, 23, 26, 31, 58,
see also Single-Root I/O Virtualization (SR-IOV)

Device Lending

One of the sharing methods of SmartIO, allowing devices to be distributed to and shared with remote machines. 14–17, 23–28, 31, 35, 37–41, 43–45, 47, 49–51, 54–58, 60

direct memory access (DMA)

Devices capable of direct memory access can access system memory and even BARs of other devices (peer-to-peer). 5, 8, 9, 16, 19–22, 25–30, 32, 35, 37, 39, 41–44, 49, 53, 56–58, 60

disaggregation

Dividing up a resource, such as memory or a device, into smaller components, and making them available to remote units. 3, 8–13, 15, 17, 23, 31, 47–52, 54–56, 58, 59, 61

DMA window

A segment mapped for a device through the lender's NTB, so that it may DMA to it. 28, 30, 32, 56,
see also shared memory segment

GPUDirect

A feature of Nvidia GPUs that supports peer-to-peer DMA to GPU memory. 13, 31, 34, 35, 49, 50,
see also peer-to-peer DMA

hot-add

A PCIe feature where a device is added to the device tree of a running system, and after the bus enumeration process. 8, 14, 27, 60

hypervisor

Kernel space software on a host machine, assisting and facilitating a virtual machine emulator (such as Qemu) with IOMMU mappings. 2, 11, 14, 28–30, 51, 54, 55,
see also I/O Memory Management Unit (IOMMU)

I/O Memory Management Unit (IOMMU)

A unit embedded on the CPU that creates separate virtual address spaces for devices and prevents DMA transactions outside these virtual address spaces. 2, 16, 22, 27–30, 37, 39, 43, 46, 54, 56, 57, 60

input/output (I/O)

Any interaction with a hardware device 1–3, 6–9, 12–16, 22, 23, 25, 26, 28–30, 32, 36, 38, 41, 44–50, 53, 54, 57, 59, 60

kernel space

Virtual memory and execution privileges suitable for OS kernel and device drivers. 61

lender

A computer system that has registered one or more of its internal devices with SmartIO and allowing it to be used by other machines. 23–25, 27–30, 33, 36, 39, 43, 44, 46, 51–58, 60,
see also borrower

Mediated Device Driver (MDEV)

One of the sharing methods of SmartIO, allowing devices to be distributed and shared with remote VMs. 14–17, 24, 25, 28–31, 37, 38, 44, 49–52, 54–58, 60,
see also pass-through

memory-mapped I/O (MMIO)

In PCIe, device memory, such as registers, are mapped to the same address space as RAM, allowing the CPU to read and write to this memory the same way it would access system memory. 20, 25, 30

message-passing

A standardized method of communication commonly used in parallel computing, including semantics for sending and receiving messages. 3, 49, 50

message-signaled interrupts (MSI)

Interrupts in PCIe are implemented as writes to a special memory address that is interpreted by the CPU to invoke the correct interrupt vector routine. 20, 21, 27

middleware

A software service that provides facilitation beyond functionality available from the OS, often implemented using RDMA. 3, 6, 8, 49, 56

multicasting

A feature supported by some PCIe switch chips that allow data coming in on one switch port to be replicated on multiple switch ports. 9, 13, 15, 31, 37, 38, 47, 58

non-transparent bridge (NTB)

A special PCIe device that translates memory transactions between separate address spaces. 4–7, 9–12, 20–30, 38–40, 43, 46, 51–56, 58, 59

paravirtualization

A paravirtualized device relies on facilitation by the hypervisor in order to use host resources. 28,
see also hypervisor

partitioned global address space

A standardized parallel programming model for distributed, shared-memory. 51

pass-through

Allowing a physical hardware device to be accessed directly by a VM guest by using a system's IOMMU to map memory for the device. 2, 14, 22, 28–30, 37, 51, 54, 56, 57, 60,
see also hypervisor

peer-to-peer DMA

A PCIe feature allowing two devices to directly transfer data between each other without going through system RAM. 8, 16, 21, 25, 28, 35, 37, 43, 58,
see also direct memory access (DMA)

remote direct memory access (RDMA)

Using the network adapter to copy memory directly onto the network, without going through the OS network stack, often used to implement middleware services. 3, 6, 8, 15, 16, 37, 38, 48–50, 56,
see also direct memory access (DMA)

remote procedure calls

A standardized method for passing messages and invoking software procedures on a remote system. 49

shadow device

The mechanism used by to intercept certain device interactions from a device driver. 26–28, 55,
see also Device Lending

shared memory segment

A contiguous range of memory that may be mapped through an NTB. Segments can be allocated in RAM or in device memory (BAR). 9, 20, 21, 24, 25, 30–33, 35, 36, 44, 46, 50, 53, 56, 58

Single-Root I/O Virtualization (SR-IOV)

Allows a single device to virtualize multiple device functions in hardware, each virtual function appearing to the system as a real device (function) with its own set of resources. 3, 9, 15, 23, 31, 37, 48–51, 58,
see also PCIe device function

trap

A hardware-assisted virtualization mechanism that makes it possible to invoke an interrupt routine with higher privileges. Also commonly known as a "fault" or "exception". 28, 29, 49

user space

Virtual memory and execution privileges suitable for application software, in contrast to kernel space. Device drivers implemented in user space by-pass the kernel for improved performance, but sacrifices functionality that requires kernel space privileges. 15, 23, 24, 30, 32, 39, 44, 55, 56, 60, 61

virtual machine guest

A computer system emulated in software. 2, 11, 14, 16, 17, 22, 28–30, 51, 54, 56, 57

virtual machine host

The physical machine running one or more VMs. 2, 10, 16, 22, 23, 26, 28–30, 51, 52, 55, 56, 60,
see also hypervisor

virtual machine emulator

The program that emulates a computer system, i.e., the VM. 22, 29, 30, 56, 60,
see also virtual machine guest

Bibliography

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajes Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Weigert. "Intel Virtualization Technology for Directed I/O." In: *Intel Technology Journal* vol. 10, no. 03 (August 2006), pp. 179–192. DOI: 10.1535/itj.1003.02 (cited on page 2).

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote Regions: A Simple Abstraction for Remote Memory." In: *Proceedings of the 24th USENIX Annual Technical Conference*. ATC'18. July 2018, pp. 775–787. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/system/files/conference/atc18/atc18-aguilera.pdf (cited on page 49).

[3] Jens Axboe. *Flexible I/O Tester*. URL: https://github.com/axboe/fio [Accessed: 04/19/2022] (cited on page 39).

[4] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. "SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs." In: *ACM Transactions on Computer Systems* vol. 36, no. 2 (July 2019), 5:1–5:26. ISSN: 0734-2071. DOI: 10.1145/3309987 (cited on page 35).

[5] Broadcom. *Multi-Host System and Intelligent I/O Design with PCI Express*. White paper. 2005. URL: https://docs.broadcom.com/docs-and-downloads/pdf/technical/expresslane/NTB_Brief_April-05.pdf [Accessed: 05/09/2022] (cited on pages 4, 20).

[6] Broadcom. *PEX8733, PCI Express Gen 3 Switch, 32 Lanes, 18 Ports*. August 2011. URL: https://docs.broadcom.com/docs/12351852 [Accessed: 02/16/2022] (cited on pages 20, 39, 51).

[7] I-Hsin Chung, Bulent Abali, and Paul Crumley. "Towards a Composable Computer System." In: *Proceedings of the 1st ACM SIGHPC International Conference on High Performance Computing in Asia-Pacific Region*. HPC Asia'18. January 2018, pp. 137–147. DOI: 10.1145/3149457.3149466 (cited on pages 3, 48).

[8] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. "Deep Learning with COTS HPC Systems." In: *Proceedings of the 30th International Conference on Machine Learning*. ICML'13. June 2013, pp. 1337–1345. URL: http://proceedings.mlr.press/v28/coates13.pdf (cited on page 1).

[9] Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. "Computing as a Discipline." In: *Communications of the ACM* vol. 32, no. 1 (January 1989). Ed. by Peter J. Denning, pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239 (cited on page 12).

[10] Peter J. Denning. "Is Computer Science Science." In: *Communications of the ACM* vol. 48, no. 4 (April 2005), pp. 27–31. ISSN: 0001-0782. DOI: 10.1145/1053291.1053309 (cited on page 11).

[11] Dolphin Interconnect Solutions. *PXH830 PCI Express Gen3 x16 NTB Host Adapter.* URL: https://www.dolphinics.com/products/PXH830.html [Accessed: 04/16/2022] (cited on page 38).

[12] José Duato, Antonio J. Pena, Frederico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. "rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters." In: *Proceedings of the 8th IEEE International Conference on High Performance Computing & Simulation.* HPCS'10. June 2010, pp. 224–231. DOI: 10.1109/HPCS.2010.5547126 (cited on page 49).

[13] Amnon H. Eden. "Three Paradigms of Computer Science." In: *Minds and Machines* vol. 17, no. 2 (July 2007), pp. 135–167. ISSN: 1572-8641. DOI: 10.1007/s11023-007-9060-8 (cited on page 11).

[14] NVM Express. *NVM Express Base Specification.* Revision 1.3d. March 2019. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3d-2019.03.20-Ratified.pdf [Accessed: 02/16/2022] (cited on page 32).

[15] NVM Express. *NVM Express over Fabrics.* Revision 1.1. October 2019. URL: https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf [Accessed: 02/16/2022] (cited on page 49).

[16] Trevor Fountain, Alexandra McCarthy, and Fangfang Peng. "PCI Express: An Overview of PCI Express, Cabled PCI Express and PXI Express." In: *Proceedings of the 10th International Conference on Accelerator & Large Experimental Physics Control Systems.* ICALEPCS'05. October 2005. URL: https://accelconf.web.cern.ch/ica05/proceedings/pdf/I3_001.pdf (cited on page 3).

[17] GigaIO. *GigaIO Rack-scale Composable Infrastructure.* URL: https://gigaio.com/ [Accessed: 05/09/2022] (cited on page 48).

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdury, and Kang G. Shin. "Efficient Memory Disaggregation with INFINISWAP." In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation.* NSDI'17. March 2017, pp. 649–667. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-gu.pdf (cited on page 49).

[19] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balkrishnan. “Performance Characterization of NVMe-over-Fabrics Storage Disaggregation.” In: *ACM Transactions on Storage* vol. 14, no. 4 (December 2018), 31:1–31:18. ISSN: 1553-3077. DOI: 10.1145/3239563 (cited on pages 3, 49).

[20] Rui Hou, Tao Jiang, Liuhang Zhang, Pengfei Qi, Jianbo Dong, Haibin Wang, Xiongli Gu, and Shujie Zhang. “Cost Effective Data Center Servers.” In: *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture*. HPCA’13. 2013, pp. 179–187. DOI: 10.1109/HPCA.2013.6522317 (cited on page 49).

[21] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi-Ur-Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. “High-Performance design of HBase with RDMA over InfiniBand.” In: *Proceedings of 26th IEEE/ACM International Parallel and Distributed Processing Symposium*. IPDPS’12. 2012, pp. 774–785. DOI: 10.1109/IPDPS.2012.74 (cited on page 3).

[22] Neo Jia and Kirti Wankhede. *VFIO Mediated Devices*. 2016. URL: https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt [Accessed: 01/21/2022] (cited on page 28).

[23] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, and Rajeev Thakur. “High performance MPI-2 one-sided communication over InfiniBand.” In: *Proceedings of the 4th IEEE International Symposium on Cluster Computing and the Grid*. CCGrid’04. 2004, pp. 531–538. DOI: 10.1109/CCGrid.2004.1336648 (cited on pages 3, 49).

[24] Kostas Katrinis, Dimitris Syrivelis, Dionisis Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, Kobi Hasharomi, Daniel Raho, Christian Pinto, Felix Espina, Sergio López-Buedo, Q. Chen, Mario Daniel Nemirovsky, Damian Roca, Hans Klos, and Tom Berends. “Rack-Scale Disaggregated cloud data centers: The dReDBox project vision.” In: *Proceedings of the 19th IEEE Conference on Design, Automation & Test in Europe*. DATE’16. March 2016, pp. 690–695. DOI: 10.3850/9783981537079_1014 (cited on page 48).

[25] Venkata Krishnan, Todd Comins, Rudy Stalzer, and David Wong. “A Case Study in I/O Disaggregation using PCI Express Advanced Switching Interconnect (ASI).” In: *Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*. HOTI’06. August 2006, pp. 15–24. DOI: 10.1109/HOTI.2006.5 (cited on page 48).

[26] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. “Swapping to remote memory over Infiniband: An approach using a high performance network block device.” In: *Proceedings of the 7th IEEE International Conference on Cluster Computing*. Cluster’05. September 2005, pp. 1–10. DOI: 10.1109/CLUSTR.2005.347050 (cited on page 49).

[27] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. “Disaggregated Memory for Expansion and Sharing in Blade Servers.” In: *Proceedings of the 36th Annual ACM SIGARCH International Symposium on Computer Architecture*. ISCA’09. June 2009, pp. 267–278. ISBN: 978-1-605585-26-0. DOI: 10.1145/1555754.1555789 (cited on page 49).

[28] Seung-Ho Lim, Ki-Woong Park, and Kwang-Ho Cha. “Developing an OpenSHMEM Model over a Switchless PCIe Non-Transparent Bridge Interface.” In: *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium Workshops*. IPDPS’19 Workshops. May 2019, pp. 593–602. DOI: 10.1109/IPDPSW.2019.00104 (cited on page 51).

[29] Linux kernel development community. *VFIO - "Virtual Function I/O"*. 2013. URL: https://www.kernel.org/doc/Documentation/vfio.txt [Accessed: 01/23/2022] (cited on page 22).

[30] Linux kernel development community. *Linux IOMMU Support*. URL: https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt [Accessed: 01/21/2022] (cited on page 22).

[31] Liqid Corporation. *Liqid Composable Infrastructure*. URL: https://www.liqid.com/ [Accessed: 05/09/2022] (cited on page 48).

[32] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. “High-Performance Design of Hadoop RPC with RDMA over InfiniBand.” In: *Proceedings of the 42nd ACM International Conference on Parallel Processing*. ICPP’13. October 2013, pp. 641–650. DOI: 10.1109/ICPP.2013.78 (cited on page 49).

[33] Josepth E. McGrath. “Dilemmatics: The Study of Research Choices and Dilemmas.” In: *American Behavioral Scientist* vol. 25, no. 2 (October 1981), pp. 179–210. ISSN: 0097-6407. DOI: 10.1177/000276428102500205 (cited on page 11).

[34] Vijay Meduri. *A Case for PCI Express as a High-Performance Cluster Interconnect*. January 2011. URL: https://www.hpcwire.com/2011/01/24/a_case_for_pci_express_as_a_high-performance_cluster_interconnect/ [Accessed: 05/15/2022] (cited on page 4).

[35] Microsemi. *Multi-Host Sharing of NVMe Drives and GPUs Using PCIe Fabrics*. Microsemi, October 2019. URL: https://www.microsemi.com/document-portal/doc_download/1244483-multi-host-sharing-of-nvme-drives-and-gpus-using-pcie [Accessed: 05/09/2022] (cited on pages 3, 48).

[36] Ben-Yehuda Muli, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, Jun Nakijima, and Elsie Wahlig. “Utilizing IOMMUs for virtualization in Linux and Xen.” In: *Proceedings of the 8th Ottawa Linux Symposium*. OLS’06. July 2006, pp. 71–85. URL: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-71-86.pdf (cited on pages 2, 22).

[37] NVIDIA Corporation. *GPUDirect RDMA Documentation.* 2022. URL: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html [Accessed: 01/26/2022] (cited on pages 13, 31, 35).

[38] NVIDIA Corporation. *CUDA Toolkit Documentation v11.6.2.* URL: http://docs.nvidia.com/cuda/ [Accessed: 04/20/2022] (cited on page 43).

[39] Peripheral Component Interconnect Special Interest Group (PCI-SIG). *Multi-Root I/O Virtualisation and Sharing Specification.* Revision 1.x. May 2008. URL: https://pcisig.com/specifications [Accessed: 02/11/2022] (cited on page 48).

[40] Peripheral Component Interconnect Special Interest Group (PCI-SIG). *Address Translation Services Revision 1.1.* Revision 1.1. January 2009. URL: https://www.pcisig.com/specifications [Accessed: 05/10/2022] (cited on page 60).

[41] Peripheral Component Interconnect Special Interest Group (PCI-SIG). *Single Root I/O Virtualisation and Sharing Specification.* Revision 3.1. January 2010. URL: https://pcisig.com/specifications [Accessed: 01/25/2022] (cited on page 31).

[42] Peripheral Component Interconnect Special Interest Group (PCI-SIG). *PCI Express Base Specification 4.0.* Version 1.0. October 2017. URL: https://pcisig.com/specifications [Accessed: 12/17/2021] (cited on pages 2, 19, 31).

[43] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J. Newburn, Dimitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. *BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage.* Version 2. March 23, 2022. DOI: 10.48550/ARXIV.2203.04910. URL: https://arxiv.org/abs/2203.04910 [Accessed: 03/24/2022] (cited on page 15).

[44] Murali Ravindran. "Extending Cabled PCI Express to Connect Devices with Independent PCI Domains." In: *Proceedings of the 2nd Annual IEEE International Systems Conference.* SysCon'08. April 2008, pp. 1–7. DOI: 10.1109/SYSTEMS.2008.4519048 (cited on pages 3, 48).

[45] Jack Regula. *Using Non-Transparent Bridging in PCI Express Systems.* White paper. Broadcom, 2004. URL: https://docs.broadcom.com/doc/12353428 [Accessed: 05/09/2022] (cited on pages 3, 4).

[46] Davide Rosetti. *Benchmarking GPUDirect RDMA on Modern Server Platforms.* NVIDIA Corporation. October 2014. URL: https://developer.nvidia.com/blog/benchmarking-gpudirect-rdma-on-modern-server-platforms/ [Accessed: 01/26/2022] (cited on pages 13, 31, 49).

[47] Nikolay Sakharnykh. *Beyond GPU Memory Limits with Unified Memory on Pascal.* December 2016. URL: https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/ [Accessed: 11/28/2021] (cited on pages 13, 50).

[48] Cheol Shim, Kwang-Ho Cha, and Min Choi. “Design and implementation of initial OpenSHMEM on PCIe NTB based Cloud Computing.” In: *Cluster Computing* vol. 22 (February 2018): *Supplement issue 1*, pp. 1815–1826. DOI: 10.1007/s10586-018-1707-0 (cited on pages 4, 51).

[49] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. “Shoal: A Network Architecture for Disaggregated Racks.” In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation.* NSDI’19. February 2019. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/system/files/nsdi19-shrivastav.pdf (cited on page 48).

[50] Mark J. Sullivan. *Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge.* White paper. Intel Corporation, January 2010 (cited on page 20).

[51] Jun Suzuki, Yoichi Hidaka, Hunichi Higuchi, Masaki Kan, and Takashi Yoshikawa. “Disaggregation and Sharing of I/O Devices in Cloud Data Centers.” In: *IEEE Transactions on Computers* vol. 65 (10 December 2016), pp. 3013–3026. DOI: 10.1109/TC.2015.2513759 (cited on page 48).

[52] Amir Taherkordi, Feroz Zahid, Yiannis Verginadis, and Geir Horn. “Future Cloud System Designs: Challenges and Research Directions.” In: *IEEE Access* vol. 6 (November 2018), pp. 74120–74150. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2883149 (cited on page 1).

[53] The CXL Consortium. *Compute Express Link: The Breakthrough CPU-to-Device Interconnect.* URL: https://www.computeexpresslink.org/ [Accessed: 05/15/2022] (cited on page 61).

[54] Adam Thompson and Chris J. Newburn. *GPUDirect Storage: A Direct Path Between Storage and GPU Memory.* NVIDIA Corporation. August 2019. URL: https://developer.nvidia.com/blog/gpudirect-storage/ [Accessed: 01/26/2022] (cited on page 35).

[55] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. “A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity.” In: *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems.* APSys’11. July 2011, 17:1–17:5. DOI: 10.1145/2103799.2103820 (cited on page 1).

[56] Shin-Yeh Tsai and Yiying Zhang. “A Double-Edged Sword: Security Threats and Opportunities in One-Sided Network Communication.” In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing.* HotCloud’19. July 2019. URL: https://www.usenix.org/system/files/hotcloud19-paper-tsai.pdf (cited on page 60).

[57] Cheng-Chun Tu. “Memory-Based Rack Area Networking.” PhD thesis. Stony Brook University, May 2014 (cited on page 51).

[58] Cheng-Chun Tu and Tzi-cker Chiueh. “Seamless Fail-over for PCIe Switched Networks.” In: *Proceedings of the 11th ACM International Systems and Storage Conference*. SYSTOR’18. June 2018, pp. 101–111. DOI: 10.1145/3211890.3211895 (cited on page 52).

[59] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. “A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery.” In: *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE’15. March 2015, pp. 1–15. ISBN: 978-1-4503-3450-1. DOI: 10.1145/2731186.2731189 (cited on page 51).

[60] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. “Secure I/O Device Sharing Among Virtual Machines on Multiple Hosts.” In: *ACM SIGARCH Computing Architecture News* vol. 41, no. 3 (June 2013), pp. 108–119. DOI: 10.1145/2508148.2485932 (cited on pages 6, 51, 55).

[61] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. “Marlin: A Memory-based Rack Area Network.” In: *Proceedings of the 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS’14. October 2014, pp. 125–136. DOI: 10.1145/2658260.2658262 (cited on pages 4, 49).

[62] Akshay Venkatesh, Khaled Hamidouche, Sreeram Potluri, Davide Rosetti, Ching-Hsiang Chu, and Dhabaleswar K. Panda. “MPI-GDS: High Performance MPI Designs with GPUDirect-aSync for CPU-GPU Control Flow Decoupling.” In: *Proceedings of the 46th ACM International Conference on Parallel Processing*. ICPP’17. August 2017, pp. 151–160. DOI: 10.1109/ICPP.2017.24 (cited on page 49).

[63] Akshay Venkatesh, Hari Subramoni, Khaled Hamidouche, and Dhabaleswar K. Panda. “A high performance broadcast design with hardware multicast and GPUDirect RDMA for streaming applications on Infiniband clusters.” In: *Proceedings of the 21st IEEE International Conference on High Performance Computing*. HiPC’14. December 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116875 (cited on pages 3, 49).

[64] Subianto Windoro. *Switch Partitioning in PCI Express Switches*. Renesas, December 2008. URL: https://www.renesas.com/eu/en/document/apn/708-switch-partitioning-pcie-switches [Accessed: 02/14/2022] (cited on pages 3, 48).

[65] Heymian Wong. “PCI Express Multi-Root Switch Reconfiguration During System Operation.” MA thesis. Massachusetts Institute of Technology, May 2011. DOI: 1721.1/66819 (cited on page 48).

[66] Xiangliang Yu. *NTB: Add support for AMD PCI-Express Non-Transparent Bridge*. January 2016. URL: https://lwn.net/Articles/672752/ [Accessed: 10/01/2021] (cited on page 20).

Published Papers

Paper I

Device Lending in PCI Express Networks

Authors: Lars Bjørlykke Kristiansen, **Jonas Markussen**, Håkon Kvale Stensland, Michael Riegler, Hugo Kohmann, Friedrich Seifert, Roy Nordstrøm, Carsten Griwodz, Pål Halvorsen.

Abstract: The challenge of scaling IO performance of multimedia systems to demands of their users has attracted much research. A lot of effort has gone into development of distributed systems that add little latency and computing overhead. For machines in PCI Express (PCIe) clusters, we propose Device Lending as a novel solution which works at a system level. Device Lending achieves low latency and extremely low computing overhead without requiring *any* application-specific distribution mechanisms. For the application, the remote IO resource appears local. In fact, even the drivers of the operating system remain unaware that hardware resources are located in remote machines. By enabling machines in a PCIe cluster to lend a wide variety of hardware, cluster machines can get temporary access to a pool of IO resources. Network cards, FPGAs, SSDs, and even GPUs can easily be shared among computers. Our proposed solution, Device Lending, works transparently without requiring any modifications to drivers, operating systems or software applications.

Candidate's contributions: Based on Kristiansen's initial implementation of Device Lending, Markussen had several discussions with Kristiansen and contributed to its development through testing and conducting performance benchmarks. Additionally, Markussen was responsible for writing most of the text and organizing the collaboration with all of the authors. He designed and performed the performance evaluation of the Device Lending method, and also implemented the GPU RDMA benchmark program used in the performance evaluation.

Published in: *Proceedings of the 26th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV'16. ACM. May 2016, article 10, pp. 10:1–10:6.

DOI: 10.1145/2910642.2910650

Contributed to: Objectives 1, 2 and 6.

Device Lending in PCI Express Networks

Lars Bjørlykke Kristiansen[1], Jonas Markussen[2], Håkon Kvale Stensland[2], Michael Riegler[2], Hugo Kohmann[1], Friedrich Seifert[1], Roy Nordstrøm[1], Carsten Griwodz[2], Pål Halvorsen[2]

[1]Dolphin Interconnect Solutions AS, Norway
[2]Simula Research Laboratory, Norway & University of Oslo, Norway

{larsk, hugo, sfr, royn}@dolphinics.no
{jonassm, haakonks, michael, griff, paalh}@simula.no

ABSTRACT

The challenge of scaling IO performance of multimedia systems to demands of their users has attracted much research. A lot of effort has gone into development of distributed systems that add little latency and computing overhead. For machines in PCI Express (PCIe) clusters, we propose Device Lending as a novel solution which works at a system level.

Device Lending achieves low latency and extremely low computing overhead without requiring *any* application-specific distribution mechanisms. For applications, the remote IO resource appears local. In fact, even the drivers of the operating system remain unaware that hardware resources are located in remote machines.

By enabling machines in a PCIe cluster to lend a wide variety of hardware, cluster machines can get temporary access to a pool of IO resources. Network cards, FPGAs, SSDs, and even GPUs can easily be shared among computers. Our proposed solution, Device Lending, works transparently without requiring any modifications to drivers, operating systems or software applications.

CCS Concepts

•Computer systems organization → Distributed architectures; •Software and its engineering → *Distributed systems organizing principles;*

Keywords

Multimedia, GPU, PCIe, interconnect, device sharing

1. INTRODUCTION

Performing multimedia tasks in real time are challenging and frequently require distributed systems. Tetzlaff et al. [28] early provided a classification for designing a distributed system. Actual implementations have often addressed requirements for low latency and high throughput by specialized interconnect networks [8, 6, 10, 7]. The PCI Express (PCIe) interconnect network [5, 19], which today is the dominant interconnection technology inside individual computers, can be connected to the internal networks of remote computers by using PCIe non-transparent bridges (NTB) [23]. The communication over such an interconnect network may be performed just like in classical interconnected networks, for example by implementing a high-performance TCP/IP stack for PCIe [11].

From the point of view of each computer, an NTB is just another PCIe device that offers memory areas for mapping into the remote computer's physical address space. An unusual property of the NTB, is that this memory is not located on it, but is rather a mapping of arbitrary memory areas within the domain of other computers that are also connected to the same NTB.

This raises the question whether all PCIe devices that are connected to any of the computers attached to such an NTB, can be considered part of one common resource pool. With Device Lending, devices can by lent by one computer into another without involving the CPU in data path forwarding.

All resources of any PCIe device are represented by mapped addresses, including their control registers and interrupts, so all of them can be mapped by an NTB. Obviously, such mapping cannot be trivial. Whereas data areas can be mapped into a computer's address space just like those of locally installed devices, a reverse mapping is required for interrupts. Furthermore, devices can be lent dynamically by one computer to another only if the operating systems can handle that PCIe devices are added to and removed from their address space, i.e., if they have hotplug support [14] for the specific device.

Once these problems are solved, we can see that the power of this approach goes far beyond the classic interconnection challenges of a streaming server. Within a small cluster, devices can be pooled together and time-shared by different

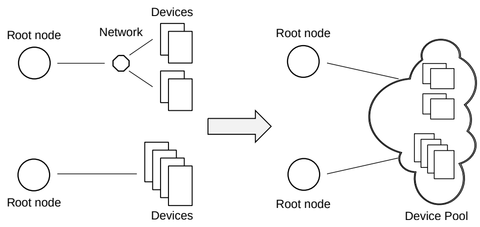


Figure 1: PCIe devices on separate machines could be pooled together and shared between multiple computers.

NOSSDAV'16, May 13 2016, Klagenfurt, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4356-5/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2910642.2910650

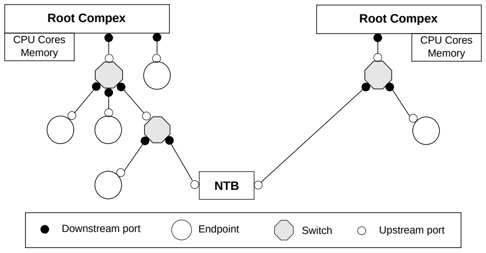


Figure 2: An example of a PCIe topology.

computers (Figure 1). Network cards can be assigned to a computer while it needs high throughput. Instead of copying data between SSD disks over traditional network, the disk can be borrowed and accessed directly. For a large CUDA programming task, a computer can lend additional cards and use CUDA's own peer-to-peer model instead of relying on additional middleware like rCUDA [4]. Pogorelov et al.[21] have shown how a multimedia workload can be offloaded to a remote GPU using Device Lending.

In this paper, we present how we achieve this pooling of PCIe devices using only native device drivers. We present the state of our proof-of-concept implementation of Device Lending for Ethernet network cards and SSD disks, and in more detail, our prototype for GPU lending. We show that the GPUs can be lent dynamically without any modifications to drivers or user-space applications.

The paper is organized as follows: we present essential capabilities of PCIe in Section 2. Section 3 addresses the current state of PCIe virtualization support. In Section 4 we discuss related work. Section 5 goes into details of our implementation of Device Lending, followed by performance results for GPU lending in Section 6. Conclusion and further opportunities are discussed in Section 7.

2. PCI EXPRESS

PCIe is an industry standard for architecture-independent connection of hardware peripherals to computers. In PCIe terminology, such a peripheral is a *PCIe endpoint*. While its predecessor PCI relied on parallel buses that were shared between endpoints, PCIe uses point-to-point links (still called *buses*) that consist of 1 to 32 *lanes*. These buses can be connected to *PCIe switches*, which may be connected to other switches, forming a tree structure where endpoints are leaves, switches are inner nodes, and buses are edges. An example of a PCIe topology is illustrated in Figure 2. The connection of a bus to a switch is called a *port*, but (primarily to illustrate how backwards compatibility with PCI is achieved) it is also known as a *bridge*. Ports towards the tree root are called *upstream*, the other *downstream*. The network of buses, endpoints and switches is referred to as *fabric*. For communication, PCIe specifies a layered protocol structure, whose upper layer is called transaction layer, exchanging transaction layer packets (TLPs). Routing occurs in a strictly hierarchical fashion, i.e., packets do not need to pass through the root of the tree.

At the root of the PCIe tree is the *root complex*, which an implementation can either interpret as an endpoint that is connected to the root node of the fabric or as being the root node. In this paper, we refer to the root complex as the root node. Directly connected to the root complex is the CPU core and memory controller. Each endpoint may act like a group of distinct devices. Each of these is called a *function* and is separately addressable by the triplet of its *bus*, *device* and *function* IDs, referred to as its *BDF*.

Both endpoints and buses are detected by reading their *configuration space*. At system boot, the *system* (BIOS or OS) scans possible BDFs for vendor IDs in a process called bus enumeration. If an endpoint or bus is present at a given BDF, the system reads the associated configuration space. This contains data structures in a standardized format [19], allowing the device to define its requirements.

2.1 Memory-mapped IO

When a configuration space is found at a given BDF, the system reads the its Base Address Registers (BARs) to determine the function's size requirements and number of address spaces that must be mapped into the host's linear address space. This mapping allows the CPU to access device registers of the endpoint through regular memory accesses. This process is called Memory Mapped IO (MMIO) and allows memory operations to be transparently translated into TLPs by devices and the CPU.

The system writes the mapped addresses into the BARs, which allows the endpoint to interact with the host machine. If the device has an onboard Direct Memory Access (DMA) engine, it can be instructed to read from and write to any memory buffers directly, including main memory and other endpoints. Without a DMA engine, the CPU must write to MMIO registers to transfer data.

2.1.1 Posted and non-posted transactions

Some PCIe requests require end-to-end notification upon completion. These requests are called *non-posted transactions*, while requests that do not require notification are *posted transactions*. A memory write request is an example of a posted transaction. The requester sends the write request along with the data and after it leaves the egress port it is no longer the responsibility of the requester. Memory read requests, on the other hand, requires explicit completion TLPs.

Non-posted requests are significantly affected by the length of a PCIe path. The longer the path, the higher the request-completion latency becomes. In addition, the number of read requests in flight is limited by how many the requester supports. The number of supported read requests in flight has an impact on read performance.

2.1.2 Transparent bridges

A switch is associated with one contiguous address range in the host address space and is aware of it. The address range is called *address window*, and spans all address ranges assigned to endpoints downstream of this switch. Each port on the root complex is associated with its own contiguous address range. This allows shortest-path routing in the tree based on physical address. Switches and their ports perform only routing in this scenario, and are *transparent* in that sense. PCIe bridges can be regarded as transparent bridges.

2.1.3 Non-transparent bridges

It is desirable to extend PCIe out of the single computer and use it for high-speed interconnection networks due to its high bandwidth and low latency [22]. One way of doing this

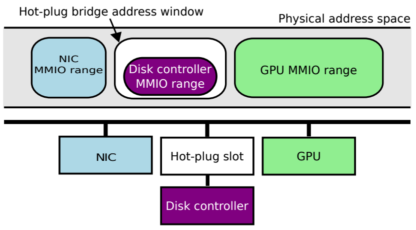


Figure 3: Physical address ranges are reserved by OS or BIOS at boot time. Memory requirements of hot-plugged devices must fit within the already existing address windows.

is by using NTBs [23]. Although not standardized, NTBs are widely adopted and all NTB implementations have similar capabilities. Several processor architectures, including recent Intel Xeon CPUs, support NTB implementations [26].

Despite the name, NTBs do actually appear as PCIe endpoints in one or more PCIe fabrics at the same time. They are mapped with large MMIO areas similar to other endpoints. However, unlike other endpoints and like transparent bridges, memory operations on these areas are forwarded from one fabric into another. Since an NTB is mapped differently in each host's address space, it performs address translation on the TLPs during forwarding. This address translation is similar to a single-level page table. Effectively, NTBs create a shared memory architecture across several hosts [13].

However, an NTB address space is not necessarily linear. Its MMIO area is divided into equally sized segments, and each segment can be mapped anywhere into the remote host's address space. This is done by replacing part of the address with a per-segment offset into the remote host's address space. Not only does this allow a remote host to access local RAM memory, it also enables a remote host to access MMIO areas of local PCIe devices.

2.2 Message-signaled interrupts

Whereas physical interrupts lines were used in traditional PCI, PCIe uses *Message-Signalled Interrupts* (MSI) [17, 19]. When an endpoint issues an MSI, this is actually a normal memory write to a special address, which is then interpreted by the chipset and used to generate an interrupt to the CPU. For our work, this has the essential implication that the address of an MSI can be mapped through an NTB.

2.3 Hot-plugging

The idea of lending devices without any OS changes whatsoever includes the goal that the devices must appear to and disappear from the OS at run-time. Obviously, there are device drivers that are not capable of coping with run-time appearance or disappearance. We can address the challenges that occur on a level "underneath" the OS.

PCIe specifies the ability of hot-plugging devices, making them available to the system while it is running. This ability was designed for replacing devices without rebooting the machine [22, 14]. Consequently, most OS implementations reserve MMIO ranges at boot time and keep them unchanged until reboot.

This is sufficient for hot-plugging in the sense of *hot-replace*, but problematic for *hot-add*, as shown in Figure 3.

When a device is hot-plugged, it appears in a port of a PCIe switch whose contiguous address range has already been mapped. A worst-case reservation for an arbitrary endpoint for every hot-plug capable port of a switch is not usual but may be feasible. However, a hot-add operation may plug an entire subtree of devices into the port, with an arbitrarily large requirement for MMIO range. If the required address range is too large, a remapping of the host address space must be undertaken. This is, however, non-trivial, and few OSes support it currently. In our implementation, the hot-add variant of hot-plugging becomes trivial, as devices become accessible through the NTB. The already allocated address space is large enough to contain all the MMIO areas.

3. VIRTUALIZATION SUPPORT IN PCIE

Traditionally, virtualization has been used to provide host resources to guest OSes in virtual machines (VM). Since endpoints are already mapped into the host address space, and the VM has a different memory layout than the host, they can traditionally not access endpoints without specialized drivers in the guest OS, which are aware of the mapping. Due to the performance penalty of this (and the breach of VM isolation that a common memory layout would bring), dedicated virtualization units have been introduced.

3.1 IO Memory Management Unit

By organizing memory in pages and adding a software-defined page-table, a Memory Management Unit (MMU) can translate addresses accessed by the CPU before passing them to chipset and memory controller. The MMU provides every processes in the host OS as well as every guest OS in a VM their own virtual, linear address space, while the physical memory can be fragmented or non-existent (e.g., swapped out).

The IO Memory Management Unit (IOMMU) [9] is similar to an MMU, but it provides virtualization of addresses between chipset (including CPU cores and MMU) and PCIe fabric. One of the most important features of the IOMMU is the DMA remapper, which translates addresses of memory operations from any IO device. In other words, it translates IO virtual addresses to physical addresses.

Similarly to pages mapped by an MMU, an IOMMU can group PCIe functions into *domains*, where each domain has separate mappings and its own address space. Such a domain can be part of the address space of a VM, while other PCIe functions remain isolated from the VM. This allows the VM to interact directly with the device using native device drivers in the guest OS, often referred to as *PCIe passthrough*.

Importantly, there is nothing that prevents the IOMMU from performing such a mapping for the host OS as well. This is an opportunity for Device Lending.

3.2 Single-Root IO Virtualization

Unlike the MMU's page maps, IOMMU mappings are not process-specific. Since IOMMU supports only one mapping per PCIe function, it can only assign an endpoint function to a single VM at a time. Single-Root IO Virtualisation (SR-IOV) [20] addresses this. SR-IOV-aware device can allow single physical PCIe functions to act as multiple virtual PCIe functions, allowing SR-IOV to map a single physical function to several VMs.

3.3 Performance penalty

As with most abstractions, DMA remapping brings a performance overhead. The translation tables are held in memory like the MMU's. When a memory access passes through, the IOMMU must perform a multi-level table look-up. Furthermore, it is located in the root complex, and all TLPs must be routed through the root to perform DMA remapping. In addition, unpredictable access patterns using small-sized pages can lead to thrashing of the IO translation look-aside buffer. PCI-SIG has developed an extension of the transaction layer protocol that allows caching of mapped addresses on the PCIe devices [19], but this is not widely available yet.

4. RELATED WORK

The idea of a unified bus for the inner components of a computer with those of another is not new. It was imagined for both ATM [24] and SCI [1]. These ideas never got implemented, because none of these technologies were picked up for the internal interconnection networks of computers.

PCIe is the dominant standard for the internal interconnection network. It is also proving to be a relevant contender for an external interconnection network. 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.

4.1 Alternative protocols

There are several interconnection technologies, which are more widely adopted for creating high-speed interconnection networks than PCIe. These include InfiniBand, as well as 10Gb Ethernet. They may achieve the same throughput on interconnection links, but they are not integrated as closely with the system fabric as PCIe, and require soft-processing of protocol stacks. Their latency is therefore, inevitably, higher than that of PCIe interconnects.

4.2 Multi-Root IO Virtualization

Multi-Root IO Virtualization (MR-IOV) [18] specifies how several hosts can be connected to the same PCIe fabric. The fabric is logically partitioned into separate virtual hierarchies, where each host sees its own hierarchy without knowing about MR-IOV. MR-IOV require multi-root aware PCIe switches, and, in the same way as SR-IOVs require SR-IOV-aware devices to provide functions to several VMs, devices must be multi-root aware to provide functions to several virtual hierarchies (and thus hosts) at the same time.

Despite being standardized in 2008 [18], we are not aware of any MR-IOV-capable devices and very few switches. Instead, there are attempts to achieve MR-IOV-like functionality through a combination of SR-IOV with NTB-like hardware [27].

4.3 Ladon and Marlin

Our Device Lending idea is apparently timely, because very similar functionality was proposed in Cheng-Chun Tu et al. in the form of the Ladon [29] and Marlin [30] systems.

Ladon uses all PCIe and virtualization features as proposed in this paper, but it achieves less freedom than our Device Lending. In Ladon, PCIe devices that are offered for sharing are all managed by a dedicated computer, the management host. The only task of the management host is to manage sharing of the devices. The guest OSes that include these devices into their PCIe fabric are, first, all running in VMs, and second, they include the remote PCIe devices in their fabric for the entire lifetime of the OS. With our Device Lending, we can actually pool the resources of a small cluster of NTB-connected devices by lending in arbitrary direction. We can even exchange devices, and do this under the control of a running OS, not a dedicated machine. By combining PCIe hot-plug support in the OS with use of the NTB, we can insert remote PCIe devices while the OS is running. Finally, for devices whose native device drivers support hot-remove, we can stop borrowing without rebooting.

Marlin [30] can share network IO capacity in a cluster by forwarding Ethernet packets underneath the host's TCP/IP stack to another node, using an Ethernet-over-PCIe driver for legacy software and a dedicated stack for zero-copy mode. While this replicates Dolphin Interconnect Solutions' (Dolphin) SuperSocket approach [12], which is a continuation of SuperSockets for SCI [25], the technique appears generic for all interconnection technologies. With Device Lending, however, we borrow the network card from the remote host and require neither driver nor encapsulation overhead.

5. IMPLEMENTATION

We have implemented Device Lending for an unmodified Linux kernel, using an NTB and the IOMMU. The implementation is composed of two parts, the *lending* side and the *borrowing* side. For our proof-of-concept implementation, we rely on a NTB implementation from Dolphin, namely the PXH810 host adapter [2].

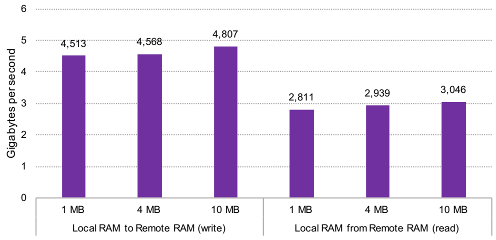
The lending side kernel module binds itself as a driver for the targeted PCIe devices. This provides us with exclusive access to the device, allowing the kernel module to access the device's configuration space while preventing other drivers on the host from interfering. The kernel module then notifies the borrowing side of all available devices.

When the user requests an available device, the borrowing side kernel module communicates with the lending side kernel module in order to read the device's configuration space. The lending side sets the targeted device into a per-borrower IOMMU domain, isolating the device from the rest of the system and other devices. The borrowing side then sets up the necessary MMIO mappings using the NTB and tells the lending side to set up the reverse mappings for device to RAM DMA as well as MSI mappings. Following this, the borrowing side then injects the device into the Linux PCI subsystem and signals a hot-add event. Linux will probe the device, set it up and load the device driver.

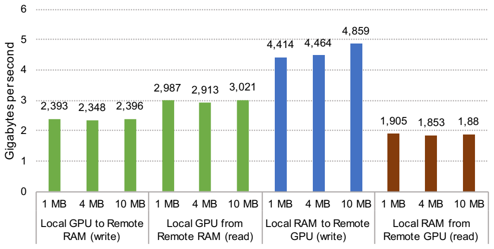
The device driver is now able to communicate with the device using MMIO access. Whenever the device driver sets up new DMA mappings using the Linux DMA-API, the borrowing side kernel module intercepts these calls and dynamically sets up and tear down the necessary IOMMU mappings. This allows the borrowing side device driver to transfer data to the remote device with no additional software overhead.

6. EVALUATION AND DISCUSSION

As the global address space feature of PCIe is unique, and since, to the best of our knowledge, no MR-IOV implementations exist, our Device Lending concept has few relevant



(a) RAM to RAM



(b) RAM to GPU

Figure 4: DMA transfer bandwidth across the NTB with different transfer sizes. The DMA engine on the NTB is used.

comparisons. Alternative solutions either require extensive virtualization support or additional protocol stacks. In order to evaluate our proof-of-concept implementation, we therefore evaluate the performance compared to what is possible to achieve with specialized use of the NTB. To establish a point of reference, we measured RAM to RAM bandwidth as this shows the maximum possible transfer rate.

We configured two test machines, shown in Figure 5. Both machines have a single Nvidia Tesla K40 directly connected to the root complex each. The machines were connected together using two x8 Gen3 Dolphin PXH810 adapter cards and an external PCIe cable. In all our tests, Machine A was used to initiate transfers.

6.1 Reference evaluation

For our RAM to RAM reference, we transferred data between the two machines over the NTB and measured the bandwidth without Device Lending (Figure 4). Here, we used Dolphin's SISCI API for programming the DMA engine on the NTB itself [3, 16]. All PCIe endpoints in our setup are connected directly to the root complex, which is why transferring between remote RAM and local RAM shows the optimal performance over the NTB (Figure 4a). RAM to remote RAM latency is approximately 573 ns.

Write requests peak at around 4.8 GB/s on our test configuration, shown on the left-hand side in Figure 4a. As mentioned in Section 2.1.1, memory read requests are affected by the distance in the PCIe hierarchy because they are *non-posted* transactions. However, there are is an additional factor that also limit the performance of read operations. PCIe defines a *maximum read request size*. This is configured by the system to ensure that the bandwidth is shared among all the devices in the hierarchy. For our test system, the maximum read request size is 512 bytes, and the TLP maximum payload size is 128 bytes. The DMA engine on the NTB handles 64 read requests in flight. As seen in Figure 4a, read requests peak at around 3 GB/s on

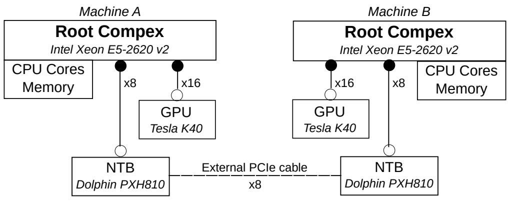


Figure 5: The setup used for our evaluation

our configuration.

Since the GPU is even further away than RAM, as illustrated in Figure 5, we see a considerably lower bandwidth for RAM to remote GPU and GPU to remote RAM transfers. Figure 4b shows the results of using the DMA engine on the NTB. The two scenarios on the left-hand side show using a local GPU on Machine A and RAM on Machine B. The two other scenarios on the right-hand side show the opposite, using local RAM on Machine A and the remote GPU on Machine B. It is important to note that when using a local GPU, the DMA engine on the NTB first has to perform read requests to the GPU before it is able to push it to the remote side using write requests. In other words, it is a two-part operation. It is interesting to note that reading from a local GPU and pushing it to remote RAM (Figure 4b, second from left) is is similar to reading from remote RAM (Figure 4a, on the right). This indicates that the latency added by the NTB is around the same as having to route TLPs through the root complex.

6.2 Device Lending evaluation

One of the novel properties of Device Lending is that it can be achieved with no modifications to endpoint devices or device drivers or even user-space software. We therefore wanted to use an already existing benchmarking tool. A well-known tool in the CUDA developer community, is the `bandwidthTest` [15] utility. This tool is included in the CUDA Toolkit samples. In default mode of operation, this program allocates page-locked buffers in RAM and measures the bandwidth it achieves when copying to the GPU and vice versa using the GPUs onboard DMA engine. We argue that making one of the most complex proprietary GPU drivers on the market work with our implementation serves as good test for our proof-of-concept.

In our setup, Machine B was configured to lend its Tesla K40 GPU to Machine A, making it available for the OS and driver on the remote machine. Figure 6 shows the results of running `bandwidthTest` on the remote Tesla K40 using different transfer sizes. The left side shows the results of making the onboard DMA engine write to remote RAM on Machine B (around 4.9 GB/s), while on the right we see the results of making the onboard DMA engine read data from remote RAM (around 2 GB/s). These numbers are comparable to the numbers seen in Figure 4b, as they show a similar scenario. However, as they use different DMA engines, they also have different locality to the data.

Using the onboard DMA engine to write to remote RAM is close to the speeds for local RAM to remote RAM trans-

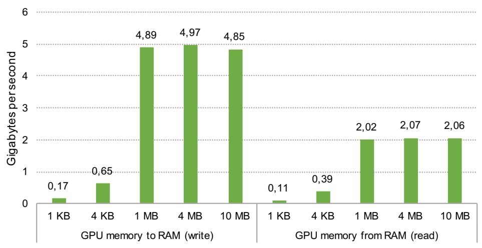


Figure 6: `bandwidthTest` running on a borrowed GPU. The DMA engine on the GPU is used to transfer.

fers (around 4.9 GB/s). Reading from remote RAM and pulling it to GPU memory (around 2 GB/s) is a bit slower than reading from remote RAM and writing it to local RAM (around 3 GB/s). This is caused by the onboard DMA engine on Machine A's GPU being even further away from the remote RAM on Machine B than the DMA engine on Machine A's NTB.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented the Device Lending concept, which allows a cluster of PCIe-connected computers to establish a pool of PCIe devices. These devices can subsequently be time-shared in a process of lending and borrowing. Since these devices appear like hot-plugged local devices to the borrowing OS, even the host OS can use them with their native drivers. For all native device drivers that support hot-plugging, these borrowed devices can be returned without rebooting. Having built the infrastructure for this, we demonstrated its performance in this paper, and provide hints for the best possible use of borrowed devices.

In further work, we will investigate concurrency challenges when multiple devices are borrowed and situations where the lender needs to take the device back forcefully. We are also planning to implement a framework for managing Device Lending. In addition, we are investigating the possibility for lending separate functions of SR-IOV devices in order to implement MR-IOV without needing specialised hardware.

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 EONS (RCN #231687) and POPART (EU H2020 #644874). The authors also acknowledge Magma for providing Nvidia Tesla GPUs.

8. REFERENCES

[1] K. Alnæs, E. H. Kristiansen, D. B. Gustavson, and D. V. James. Scalable coherent interface. In *Proc. of CompEuro*, pages 446–453, 1990.

[2] Dolphin Interconnect Solutions. PXH810 Gen3 PCI Express NTB Host Adapter.

[3] Dolphin Interconnect Solutions. SISCI API.

[4] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. of HPCS*, pages 224–231, 2010.

[5] T. Fountain, A. McCarthy, and F. Peng. PCI express: An overview of PCI express, cabled PCI express and PXI express. In *Proc. of ICALEPCS*, 2005.

[6] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and T.-W. Li. Mitra: A scalable continuous media server. *Springer Multimedia Tools and Applications*, 5(1):79–108, 1997.

[7] R. S. Grover, Q. Li, and H.-P. Dommel. Performance study of data layout schemes for a SAN-based video server. *Parallel Computing*, 34(12):747–756, 2008.

[8] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer. Architecture of a hypercube supercomputer. In *Proc. of ICPP*, pages 653–660, 1986.

[9] Intel Corporation. *Intel Virtualization Technology for Directed I/O*, 2014.

[10] T. Jones, A. Koniges, and R. Yates. Performance of the IBM general parallel file system. In *Proc. of IPDPS*, pages 673–681, 2000.

[11] V. Krishnan. Evaluation of an Integrated PCI Express IO Expansion and Clustering Fabric. In *Proc. of HOTI*, pages 93–100, 2008.

[12] V. Krishnan, T. Comins, R. Stalzer, and D. Wong. A case study in I/O disaggregation using PCI express advanced switching interconnect (ASI). In *Proc. of HOTI*, pages 15–24, 2006.

[13] L. B. Kristiansen. PCIe Device Lending: Using Non-Transparent Bridges to Share Devices. Master's thesis, University of Oslo, 2015.

[14] G. Kroah-Hartman. How the PCI hot plug driver filesystem works. *The Linux Journal*, 97(2), May 2002.

[15] NVIDIA Corporation. *CUDA Toolkit Documentation 7.5*, 2015.

[16] NVIDIA Corporation. *GPUDirect Technology Overview*, 2015.

[17] PCI-SIG. *PCI Local Bus Specification*, 2002.

[18] PCI-SIG. *Multi-root I/O Virtualization and Sharing Specification*, 2008.

[19] PCI-SIG. *PCI Express 3.1 Base Specification*, 2010.

[20] PCI-SIG. *Single-root I/O Virtualization and Sharing Specification*, 2010.

[21] K. Pogorelov, M. Riegler, J. Markussen, M. Lux, H. K. Stensland, T. Lange, C. Griwodz, P. Halvorsen, D. Johansen, P. T Schmidt, and S. L. Eskeland. Efficient processing of videos in a multi auditory environment using Device Lending of GPUs. In *In Proc. of MMSys*. ACM, 2016.

[22] M. Ravindran. Extending Cabled PCI Express to Connect Devices with Independent PCI Domains. In *Proc. of IEEE Systems Conference*, pages 1–7, 2008.

[23] J. Regula. *Using Non-transparent Bridging in PCI Express Systems*. PLX Technology, Inc, 2004.

[24] K. Saito, K. Anai, K. Igarashi, T. Nishikawa, R. Himeno, and K. Yoguchi. ATM bus system. US 5,796,741 A, 1998. US patent.

[25] F. Seifert and H. Kohmann. SCI SOCKET - a fast socket implementation over SCI. 2006.

[26] M. J. Sullivan. Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge. Technical report, 2010.

[27] J. Suzuki, Y. Hidaka, J. Higuchi, T. Baba, N. Kami, and T. Yoshikawa. Multi-root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In *Proc. of HOTI*, pages 25–31, 2010.

[28] W. Tetzlaff, M. Kienzle, and D. Sitaram. A methodology for evaluating storage systems in distributed and hierarchical video servers. In *Compcon Spring, Digest of Papers.*, pages 430–439, 1994.

[29] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh. Secure I/O device sharing among virtual machines on multiple hosts. *SIGARCH Comp. Arch. News*, 41(3):108–119, 2013.

[30] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh. Marlin: A memory-based rack area network. In *Proc. of ANCS*, pages 125–136, 2014.

Paper II

Efficient Processing of Videos in a Multi-auditory Environment using Device Lending of GPUs

Authors: Konstantin Pogorelov, Michael Riegler, **Jonas Markussen**, Håkon Kvale Stensland, Pål Halvorsen, Carsten Griwodz, Sigrun Losada Eskeland, Thomas de Lange.

Abstract: In this paper, we present a demo that utilizes Device Lending via PCI Express (PCIe) in the context of a multi-auditory environment. Device Lending is a transparent, low-latency cross-machine PCIe device sharing mechanism without any the need for implementing application-specific distribution mechanisms. As workload, we use a computer-aided diagnosis system that is used to automatically find polyps and mark them for medical doctors during a colonoscopy. We choose this scenario because one of the main requirements is to perform the analysis in real-time. The demonstration consists of a setup of two computers that demonstrates how Device Lending can be used to improve performance, as well as its effect of providing the performance needed for real-time feedback. We also present a performance evaluation that shows its real-time capabilities of it.

Candidate's contributions: Markussen discussed and developed the idea for the paper together with Pogorelov and Riegler, where Markussen was responsible for the Device Lending setup and experiments. As a demonstration of a medical computational workload utilizing Device Lending, Markussen performed the performance experiment together with Pogorelov. Markussen also contributed with text in all sections, and wrote the section on Device Lending.

Published in: *Proceedings of the 7th International Conference on Multimedia Systems.* MMSys'16. ACM. May 2016, article 36, pp. 381–386.

DOI: 10.1145/2910017.2910636

Contributed to: Objective 6.

Efficient Processing of Videos in a Multi-Auditory Environment Using Device Lending of GPUs

Konstantin Pogorelov[1], Michael Riegler[1], Jonas Markussen[1], Håkon Kvale Stensland[1]
Pål Halvorsen[1], Carsten Griwodz[1], Sigrun Losada Eskeland[3], Thomas de Lange[23]

[1]Simula Research Laboratory and University of Oslo
[2]Cancer Registry of Norwayy [3]Vestre Viken Hospital Trust

konstantin@simula.no

ABSTRACT

In this paper, we present a demo that utilizes Device Lending via PCI Express (PCIe) in the context of a multi-auditory environment. Device Lending is a transparent, low-latency cross-machine PCIe device sharing mechanism without *any* the need for implementing application-specific distribution mechanisms. As workload, we use a computer-aided diagnosis system that is used to automatically find polyps and mark them for medical doctors during a colonoscopy. We choose this scenario because one of the main requirements is to perform the analysis in real-time. The demonstration consists of a setup of two computers that demonstrates how Device Lending can be used to improve performance, as well as its effect of providing the performance needed for real-time feedback. We also present a performance evaluation that shows its real-time capabilities of it.

CCS Concepts

•Information systems → Information retrieval; Multimedia and multimodal retrieval;

Keywords

Medical Multimedia; Information Systems; Classification

1. INTRODUCTION

Colonoscopy is a medical procedure, during which specialists in bowel diseases (gastroenterologists), investigate and operate on the colon through minimally invasive surgery by using flexible endoscopes. These examinations are usually done in a special examination room as depicted in figure 1(a). A standard hospital normally has several of these rooms in their gastroenterology department. These rooms contain screens for the doctors that show the video stream from the camera, a bed for the patient, the endoscopic processor, a desktop computer for reporting and some medical

(a) The examination room where the endoscopies are performed. A usuall hospital has several of these rooms.
(b) Different endoscopes for different examinations and patients. For example the very small one is for children.
(c) The tip of the endoscope. It is very flexible and can be moved by the gastroentologist in every possible direction.
(d) The controll unit of the endoscope the gastroentologiest uses to controll the endoscope in terms of zoom, rotation, etc.

Figure 1: These images show an auditorium and endoscopic equipment in the Bærum Hospital in Norway where our system will be used.

treatment supplies. The endoscopes can vary in their attributes like the thickness of the endoscope or its length, but also in the resolution of the videos. Figure 1(b) shows a collection of different endoscopes. Endoscopes are frequently moved between examination rooms to fit the requirements of a specific examination. From the tip of the endoscope (figure 1(c)), a video is transmitted, and the gastroenterologist relies on the video stream to diagnose disease and apply treatments. To control the endoscope, the control unit that is part of every endoscope is used. As one can see in figure 1(d), this is a complex mechanism that requires a lot of concentration from the doctor during the whole procedure, lasting up to 2 hours depending on the findings. The camera can be seen as the virtual the eye of the gatroentologists, and the video stream is all they perceive. Usually, doctors get "third eye" support from their nurses to support them during the examinations and increase the number of findings.

Recently, computer-aided diagnostic systems are more and more used in gastroenterology. The most recent and best

MMSys'16 May 10-13, 2016, Klagenfurt, Austria
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4297-1/16/05.
DOI: http://dx.doi.org/10.1145/2910017.2910636

working system is Polyp-Alert [10]. This computer-aided diagnostic system helps to determine the quality of the colonoscopy during the procedure. It reaches very high accuracy and sensitivity, but it only reaches near real-time and not full real-time feedback. This is not optimal for live examinations where the medical expert controls the camera manually and cannot rely on a system that introduces delays. Even though real-time performance can be reached by using multiple GPUs in one sufficiently powerful desktop machine, placing such noisy and costly machines in the examination rooms of a hospital is impractical. A more realistic scenario is therefore to have or to use already installed smaller machines in each room and to use Device Lending whenever more resources are needed. Here, Device Lending is a concept where computers interconnected in a PCI Express network can share devices using a transparent cross-machine device sharing system without any special effors to use remote resources locally. It is a low-latency, high-throughput solution for distributed computing, utilizing common hardware already present in all modern computers and requiring little additional interconnection hardware.

In this paper, we will present a demo that utilizes Device Lending of GPUs in combination with our own computer-aided diagnosis system. With this demo, we address two main challenges. First, we will show that real-time support is possible using this technology. Second, we demonstrate the possibility of having one mainframe that can lend the devices to different computers based on the computational demands. This can be an important advantage and even required for scenarios where no room for large machines exists. Further, it can be important for setups where the requirements change fast and often on the fly (e.g., an examination room in a hospital changes the used endoscopes several times during the day; endoscopes with a very high resolution need more processing power than those with lower resolution).

2. REAL-TIME COMPUTER AIDED DIAGNOSIS SUPPORT

Automatic detection of polyps in colonoscopies has been in focus of research for a long time [9]. However, few complete systems exist that are able to do real-time detection, or that can support endoscopists by computer-aided diagnosis for colonoscopies in real-time and at the same time maintain a high detection accuracy. The most recent and best working approach is Polyp-Alert [10] that is able to give near real-time feedback during colonoscopies. Visual features and a rule based classifier are used to detect the edges of polyps, and a performance of 97.7% correctly detected polyps is reported. However, real-time support is limited as they reach only 10 frames per second.

To target the real-time performance, we have proposed EIR [8, 7, 6] medical experts supporting system for the task of detecting diseases and anatomical landmarks in the gastrointestinal (GI) tract, which used in this demo as a use case. It has several key attributes, i.e., EIR (i) is easy to use, (ii) is easy to extend to different diseases, (iii) can do real time handling of multimedia content, (iv) is able to be used as a live system and (v) has high classification performance with minimal false negative classification results. Compared to Polyp-Alert, our detection accuracy is slightly below. The classification performance of the polyp detection in our EIR system lies around a precision of 0.903 and a recall of 0.919, but it is tested on a different dataset, meaning that the numbers are not directly comparable.

Currently, the system consists of two parts, the detection subsystem that detects irregularities in video frames and images and the localisation subsystem that localises the exact position of the disease. The detection can not determine the location of the found irregularity. The location determination is done by the localisation subsystem. The localisation subsystem uses the output of the detection system as input. After the automatic detection and analysis of the content, the output has to be presented in a meaningful way to the gastroentologists. Therefore, the system has a visualisation subsystem that is reliable, robust and easy to understand also under stressful situations that can occur during a live examination. Moreover, it supports easy search and browsing through a large amount of data after the examination. In this demo, we do not focus on EIR but rather using Device Lending and how it can improve performance. EIR itself is just a relevant use case.

2.1 GPU Implementation

Parts of EIR had to be improved and changed to run on multiple GPUs and allow the system to perform in real-time. Therefore, the most compute-intensive parts have been ported to CUDA, a computation support framework for nVidia graphic cards. To achieve this, parts of the system had to be built as a heterogeneous processing subsystem. The GPU framework supports at the moment a number of features, namely Joint Composite Descriptor (JCD), which includes Fuzzy Color and Texture Histogram (FCTH) and Color and Edge Directivity Descriptor (CEDD), and Tamura, but we are working on increasing the supported features.

A main processing application interacts with a modular image processing subsystem. Both of these are implemented in Java. A multi-threading architecture is used by the image processing unit to handle multiple processing and feature extraction requests at the same time. A shared library that is responsible for maintaining connection with and stream data to the stand-alone CUDA-enabled processing server is implemented in C++. To ensure high data transfer performance and reduce excessive data copy operations, shared memory has been used, while sending requests and receiving status responses uses local UNIX sockets. A CUDA server implemented in C++ runs in the background and performs computations on GPU. The whole system can easily be extended with multiple CUDA servers running locally or on a number of remote servers. This is also valid for the processing server, which can be extended with new feature extractors and advanced image processing algorithms, and utilize multi-core CPU and GPU resources concurrently.

2.2 Device Lending

Device Lending is a concept where computers interconnected in a PCI Express [5] network can share devices. It provides transparent, low-latency cross-machine PCIe device sharing without *any* need to implement application-specific distribution mechanisms or modify native device drivers. As the workload increases or decreases, the system can allocate and de-allocate additional resources.

Today, PCIe is the most common interconnection network inside a computer, and with PCIe non-transparent bridges (NTB) [1], it can be turned into an interconnection network

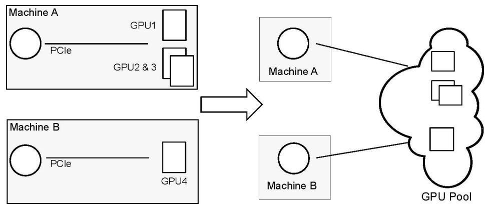


Figure 2: Pooling of devices attached in the PCIe network in the experimental setup.

Device	Type	E1	E2	E3	E4
GPU1	Nvidia Tesla K40c	*	*	*	*
GPU2	Nvidia Quadro K2200		*	*	*
GPU3	Nvidia GeForce GTX 750			*	*
GPU4	Nvidia Tesla K40c				*

Table 1: This table shows the used hardware combinations of the different experiments. GPU 1 to 3 are local GPUs. GPU4 is lend via Device Lending.

for multiple machines. In PCIe, all devices connected to the computer are considered part of one common resource pool (figure 2). All devices resources in PCIe are represented by addresses that can be mapped into a remote memory space by an NTB. Device Lending is implemented [3] using Dolphin Interconnect Solutions NTB software [1].

For the EIR system, Device Lending enables the combination of multiple GPUs through CUDA's own peer-to-peer communication model, instead of either writing a distributed system, using rCUDA [2] or MPI [4].

2.3 Performance Evaluation

To evaluate the performance of our system and also to show that Device Lending in our scenario works as intended, we performed 4 different experiment sets. An overview of the hardware used and the performed experiments can be found in table 1. For all configurations, we used the same CPU (Intel Core i7-4820K 3.7GHz) and RAM (16GB Quad Channel DDR3). The test setup consists of 2 computers (Machine A and B, see figure 2), where the host code of the tests runs on one of them. The second one lends a GPU to it. Experiment E1 uses one local GPU, E2 uses two local GPUs and E3 uses three local GPUs. In E4, we borrowed one GPU from the second computer in addition to three local GPUs. With the current machine setup it is not possible to lend more that one GPU because of software limitations in the motherboard's BIOS.

In the experiments, we performed polyp classification and real-time feedback on the video for up to 16 parallel video streams. All video streams are full HD (1920x1080) videos from colonoscopies. We measured the performance from capturing the video up to showing the output on the screen. The complete evaluation is shown in figure 3.

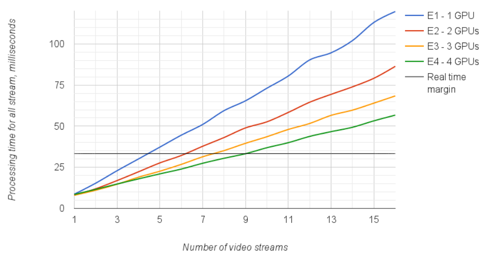
Figure 3(a) shows the performance in terms of processing time per frame for all streams simultaneous. The results reveal that for up to 7 parallel full HD streams, the 3 local GPUs are fast enough. For more than 7 streams, GPU lending is required. The graph shows that the more parallel streams are processed, the better is the performance gain from the borrowed GPU. We assume that this is due to the excessive overhead for transferring small amount of data, which hinders Device Lending to reach its full potential. This becomes less important when we have more parallel streams, and that Device Lending can indeed improve performance.

The plot in figure 3(b) shows the overall system performance. The evaluation shows that Device Lending can indeed improve the system performance. The maximum overall frames per second we reach when using 4 GPUs at the same time is 30 fps for 9 parallel full HD streams, which is equivalent to 270 fps for a single video stream. Further, this graph shows that the borrowed GPU does not increase the performance for a smaller number of videos, but for 5 and more videos the increase is higher. This is another indicator that Device Lending can increase performance a lot for large scale processing.

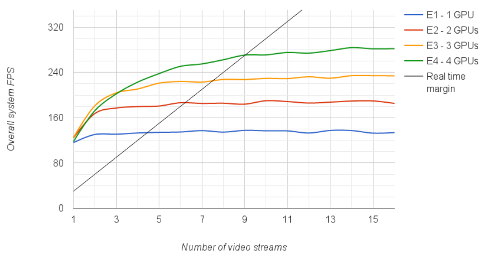
All in all, the experiments showed two important things: (i) Device Lending does not make sense for small amounts of data, but if the data to process is large it can give a large performance boost, and (ii) Device Lending makes sense in a multi-auditory scenario like we present with our demo.

3. DEMONSTRATION SETUP

The above experiments show the performance of EIR on powerful machines and that Device Lending works efficiently, i.e., high performance and low latencies at a very low overhead. However, placing such a setup in the many examination rooms in a hospital is impractical for a number of reasons like high costs and noisy machines. A more realistic scenario is therefore to have smaller machines in each room and use Device Lending whenever more resources are needed.



(a) Frame processing time for several full HD streams in parallel.



(b) Overall system performance for multiple full HD steams in parallel.

Figure 3: System performance evaluation in terms of processing time per frame and maximum performance using 4 different configurations described in table 1. Each video stream is a full HD video.

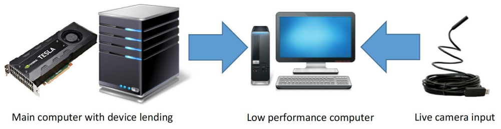


Figure 4: A compete overview of the demo setup. The demo consists of 2 computers, 1 Dolphin interconnect device, 1 screen, an artificial colon and a flexible camera. The users can use the camera in the flexible colon and will get real-time feedback about possible findings. Furthermore, the demo can be switched between Device Lending on and off to demonstrate the effect of it more clear.

To demonstrate the usefulness of Device Lending, we therefore use the above scenario. In the demo, users can use a flexible camera to perform a colonoscopy in an artificial colon, and the system will support them in real-time with analysis and feedback. The complete demo setup is depicted in figure 4. During the demo, the camera can be used to examine the artificial colon and the output of the system will be shown in real-time on the screen. The demo will show the performance increase when a GPU can is borrowed from another machine. Therefore, the demo application can be switched between lending and not lending a GPU. An example of the output for detected polyps can be seen in figure 5. This setup is similar to our real world setup of the system for live colonoscopy with videos as shown to the doctors. Thus, the processing will be done on a very weak computer that is not able to perform the complicated analysis in real-time. Therefore, it is connected to another PC via a Dolphin interconnect device and uses Device Lending to allocate the required processing power. The demo will clearly show the visible differences when Device Lending is used and when not. We also would like to point out, that the presented demonstration is based on the findings in [3] which describes the Device Lending in more detail for further reading.

4. CONCLUSION AND FUTURE WORK

In this paper, we presented a demo for Device Lending for computer-aided diagnosis that can assist medical doctors to analyse colonoscopy videos in a multi-auditory scenario. We proved that we can reach high performance in terms of processing time for several full HD video streams in parallel which make it possible to use the proposed system during several and parallel live colonoscopies. We showed that running multiple classifiers in parallel by offloading the processing to multiple machines connected through a PCI Express network and using GPU lending works in our scenario. This optimized version of the application will be able to dynamically allocate, distribute and release compute resources on demand from a pool of available GPUs. For future work, we would like to improve the scheduling of tasks within our lending network. This would include decisions for what and how much to lend to which part of the system using different input information like the required support level of doctors and the endoscope used. We also think that this idea is applicable to other scenarios like for example in cinemas where a less powerful PC in each saloon allocates GPUs based on the quality of the movie to show, e.g., one room shows 4k, one 3D and another one full HD.

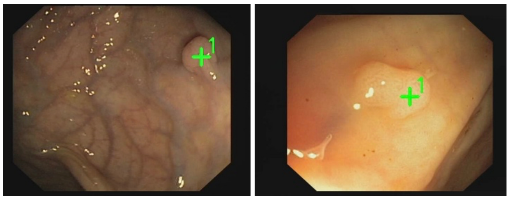


Figure 5: This figure shows 2 examples of what the doctor will see on the screen and what we will show during the demo. In both pictures, the system detected polyps and marked them with a cross. If nothing is detected, the corners of the screen are marked green for feedback.

5. ACKNOWLEDGMENT

This work has been performed in context of the FRINATEK project *EONS* (#231687) and the BIA project *PCIe* (#235530) funded by the Research Council of Norway (RCN). The authors also acknowledge Lars Bjørlykke Kristiansen and Dolphin Interconnect Solutions for assistance with Device Lending and PCIe interconnect equipment. We also would like to thank Mathias Lux from the University of Klagenfurt for “lending“ us hardware at the conference venue.

6. REFERENCES

[1] Dolphin Interconnect Solution PXH810 NTB Adapter, 2015.

[2] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. of HPCS*, pages 224–231, 2010.

[3] L. B. Kristiansen, J. Markussen, H. K. Stensland, M. Riegler, H. Kohmann, F. Seifert, R. Nordstrøm, C. Griwodz, and P. Halvorsen. Device lending in PCI Express Networks. In *Proc. of NOSSDAV*, 2016.

[4] NVIDIA Corporation. *Developing a Linux Kernel Module using GPUDirect RDMA*, 2015.

[5] PCI-SIG. *PCI Express 3.1 Base Specification*, 2010.

[6] K. Pogorelov, M. Riegler, P. Halvorsen, P. T. Schmidt, C. Griwodz, D. Johansen, S. L. Eskeland, and T. de Lange. GPU-accelerated real-time gastrointestinal diseases detection. In *Proc. of CBMS*, 2016.

[7] M. Riegler, K. Pogorelov, P. Halvorsen, T. de Lange, C. Griwodz, P. T. Schmidt, S. L. Eskeland, and D. Johansen. EIR - efficient computer aided diagnosis framework for gastrointestinal endoscopies. In *Proc. of CBMI*, 2016.

[8] M. Riegler, K. Pogorelov, J. Markussen, M. Lux, H. K. Stensland, T. de Lange, C. Griwodz, P. Halvorsen, D. Johansen, P. T. Schmidt, and S. L. Eskeland. Computer aided disease detection system for gastrointestinal examinations. In *Proc. of MMSys*, 2016.

[9] Y. Wang, W. Tavanapong, J. Wong, J. Oh, and P. C. de Groen. Near real-time retroflexion detection in colonoscopy. *IEEE BMHI*, 17(1):143–152, 2013.

[10] Y. Wang, W. Tavanapong, J. Wong, J. H. Oh, and P. C. de Groen. Polyp-alert: Near real-time feedback during colonoscopy. *CMPBM*, 120(3):164–179, 2015.

Paper III

Flexible Device Sharing in PCIe Clusters using Device Lending

Authors: **Jonas Markussen**, Lars Bjørlykke Kristiansen, Håkon Kvale Stensland, Friedrich Seifert, Carsten Griwodz, Pål Halvorsen.

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.

Candidate's contributions: Markussen came up with the idea for, designed and implemented the MDEV/KVM hypervisor extension to Device Lending. Markussen also implemented the mechanism for facilitating peer-to-peer between devices in different lenders, including the mechanism for resolving I/O addresses. Markussen identified performance issues with the original Device Lending implementation, and he contributed to investigating several solutions for improving the data path performance. Markussen wrote most of the text, and he designed and performed all the experiments, including writing the necessary performance benchmarking programs.

Published in: *Proceedings of the 47th International Conference on Parallel Processing Companion.* ICPP'18 Comp. ACM. August 2018, article 48, pp. 48:1–48:10.

DOI: 10.1145/3229710.3229759

Contributed to: Objectives 1 to 4 and 6.

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.

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.

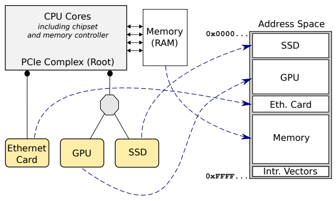


Figure 1: Device memory is mapped into the same address space as the CPUs, allowing devices to access both system memory and other devices.

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, 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.

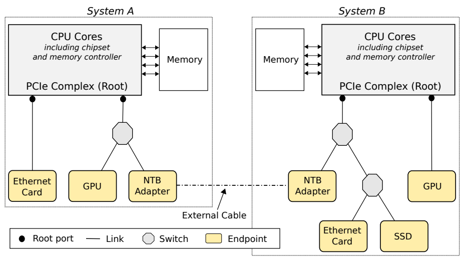


Figure 2: Example of a PCIe topology. Two independent networks are connected together using an NTB. The NTB translates IO addresses between the two different address spaces, creating a shared address space between the networks.

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.

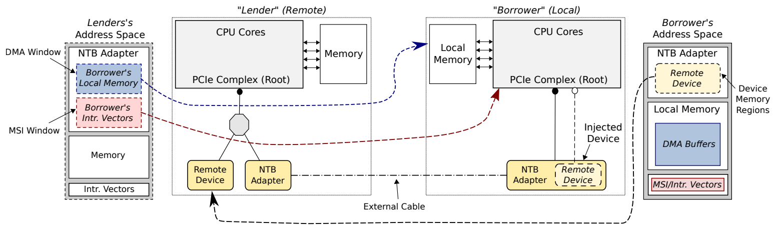


Figure 3: Using an NTB, it is possible to map the memory regions of a remote device so local CPUs are able to read and write to device registers. The remote system can in turn reverse-map the local system's memory and CPUs for the device, making DMA and MSI possible. Device Lending injects a hot-added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

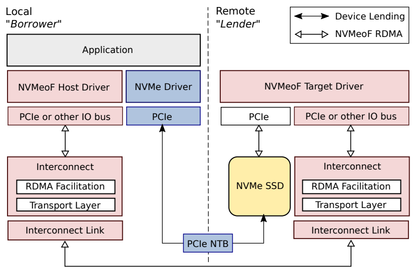


Figure 4: Illustration of native NVMe using Device Lending compared to NVMe over Fabrics using RDMA. Device Lending makes remote devices appear as if they are locally installed and there is no need for specialized support in devices or drivers.

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 hotplug 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.

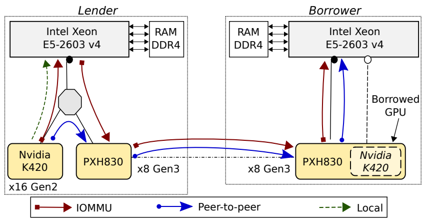


Figure 5: Configuration used in our IOMMU evaluation. The borrower is using the remote GPU. When the lender-side IOMMU is enabled, TLPs are routed through the lender's root before going over the NTB. We have also compared with a local instance, running on the lender itself.

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

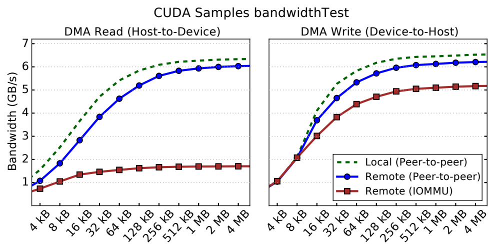


Figure 6: Reported bandwidth for different transfer sizes.

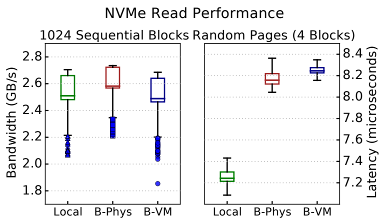


Figure 7: Bandwidth and latency when reading from disk (DMA write). We read 1024 sequential blocks for measuring bandwidth, and 4 blocks with a random offset for latency.

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

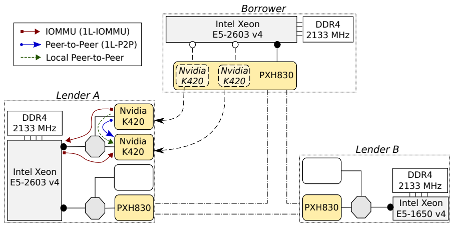
We have evaluated our KVM implementation using an Intel Optane 900P NVMe disk on a local machine without using Device Lending, a physical borrower (B-Phys), and from a VM guest (B-VM). The machines are connected back-to-back using PXH830 NTB adapters [8]. The RAM-to-RAM latency was measured to 550-580 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.

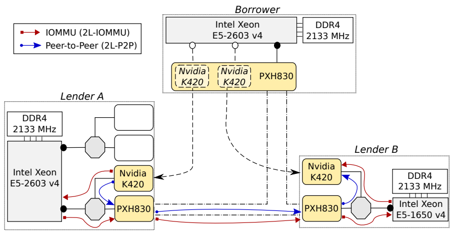
Latency was measured by reading 4 blocks repeated 10,000 times, each time at a random offset. Here, we observe that the difference between running locally and on the physical borrower is an increase in a little less than 1 microsecond. As the device now sits remotely, it has to first reach over the NTB 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,

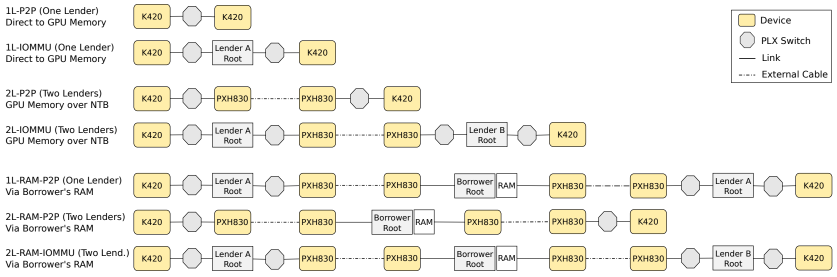


Figure 9: Data paths for the different scenarios. Each hop slightly increases the completion latency.

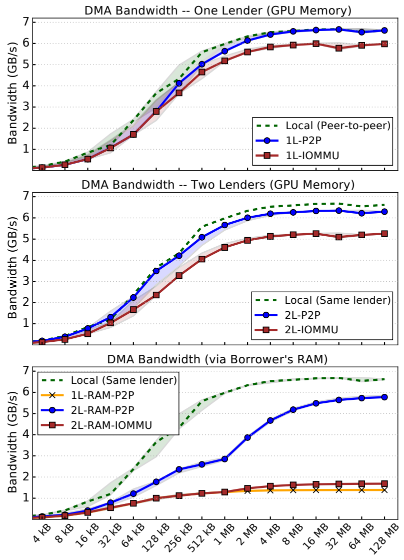


Figure 10: Median DMA bandwidth for different transfer sizes. The filled-out area represents the distribution between the 1st and 99th percentile for 10,000 runs. The local comparison is included in all three plots.

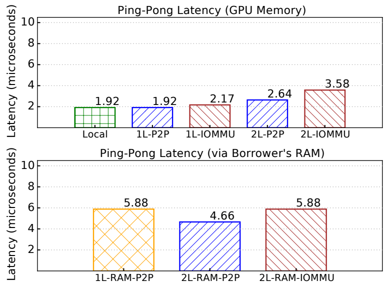


Figure 11: 99th percentiles of ping-pong latencies.

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.

Paper IV

Flexible Device Compositions and Dynamic Resource Sharing in PCIe Interconnected Clusters using Device Lending

Authors: **Jonas Markussen**, Lars Bjørlykke Kristiansen, Rune Johan Borgli, Håkon Kvale Stensland, Friedrich Seifert, Michael Riegler, Carsten Griwodz, Pål Halvorsen.

Abstract: Modern workloads often exceed the processing and I/O capabilities provided by resource virtualization, requiring direct access to the physical hardware in order to reduce latency and computing overhead. For computers interconnected in a cluster, access to remote hardware resources often requires facilitation both in hardware and specialized drivers with virtualization support. This limits the availability of resources to specific devices and drivers that are supported by the virtualization technology being used, as well as what the interconnection technology supports. For PCI Express (PCIe) clusters, we have previously proposed Device Lending as a solution for enabling direct low latency access to remote devices. The method has extremely low computing overhead, and does not require any application- or device-specific distribution mechanisms. Any PCIe device, such as network cards disks, and GPUs, can easily be shared among the connected hosts. In this work, we have extended our solution with support for a Virtual Machine (VM) hypervisor. Physical remote devices can be "passed through" to VM guests, enabling direct access to physical resources while still retaining the flexibility of virtualization. Additionally, we have also implemented multi-device support, enabling shortest-path peer-to-peer transfers between remote devices residing in different hosts. Our experimental results prove that multiple remote devices can be used, achieving bandwidth and latency close to native PCIe, and without requiring any additional support in device drivers. I/O intensive workloads run seamlessly using both local and remote resources. With our added VM and multi-device support, Device Lending offers highly customizable configurations of remote devices that can be dynamically reassigned and shared to optimize resource utilization, thus enabling a

flexible composable I/O infrastructure for VMs as well as bare-metal machines.

Candidate's contributions: This paper is an extension of Paper III. Markussen and Kristiansen developed the idea for a mechanism for detecting the guest-physical memory layout of a VM, and Markussen extended the MDEV implementation with support for this. He also extended the evaluation by designing and performing additional peer-to-peer and VM performance experiments, as well as conducting a new experiment with an I/O-heavy machine learning workload to demonstrate the improved performance. Markussen took the initiative to continue to investigate potential performance bottlenecks, and contributed to further improve the overall performance for both Device Lending and MDEV. Markussen wrote most of the text for this paper, and also improved and extended the GPU benchmarking programs used for Paper III.

Published in: *Cluster Computing.* Springer. Published online September 2019, issue date June 2020, volume 23, issue 2, pp. 1211–1234.

DOI: 10.1007/s10586-019-02988-0

Contributed to: Objectives 1 to 4 and 6.

Cluster Computing (2020) 23:1211–1234
https://doi.org/10.1007/s10586-019-02988-0

Flexible device compositions and dynamic resource sharing in PCIe interconnected clusters using Device Lending

Jonas Markussen[1,2] · Lars Bjørlykke Kristiansen[1] · Rune Johan Borgli[2,3] · Håkon Kvale Stensland[2,3] · Friedrich Seifert[1] · Michael Riegler[3,4] · Carsten Griwodz[2,3] · Pål Halvorsen[3,4]

Received: 29 March 2019 / Revised: 9 July 2019 / Accepted: 16 September 2019 / Published online: 21 September 2019

Abstract
Modern workloads often exceed the processing and I/O capabilities provided by resource virtualization, requiring direct access to the physical hardware in order to reduce latency and computing overhead. For computers interconnected in a cluser, access to remote hardware resources often requires facilitation both in hardware and specialized drivers with virtualization support. This limits the availability of resources to specific devices and drivers that are supported by the virtualization technology being used, as well as what the interconnection technology supports. For PCI Express (PCIe) clusters, we have previously proposed Device Lending as a solution for enabling direct low latency access to remote devices. The method has extremely low computing overhead, and does not require any application- or device-specific distribution mechanisms. Any PCIe device, such as network cards disks, and GPUs, can easily be shared among the connected hosts. In this work, we have extended our solution with support for a virtual machine (VM) hypervisor. Physical remote devices can be "passed through" to VM guests, enabling direct access to physical resources while still retaining the flexibility of virtualization. Additionally, we have also implemented multi-device support, enabling shortest-path peer-to-peer transfers between remote devices residing in different hosts.Our experimental results prove that multiple remote devices can be used, achieving bandwidth and latency close to native PCIe, and without requiring any additional support in device drivers. I/O intensive workloads run seamlessly using both local and remote resources. With our added VM and multi-device support, Device Lending offers highly customizable configurations of remote devices that can be dynamically reassigned and shared to optimize resource utilization, thus enabling a flexible composable I/O infrastructure for VMs as well as bare-metal machines.

Keywords Resource sharing · KVM · Composable infrastructure · Virtual machines · PCIe · Non-transparent bridging

1 Introduction

The demand for processing power and I/O resources in a cluster may, to a large degree, vary over time. Workloads come and go, and even vary themselves with number of users and amount of data to process. In this respect, efficient and dynamic resource sharing and configuration is important as it is desirable to be able to scale up and allocate more resources on demand, or scale down and release them when the resources are no longer needed. Dynamically scaling up or down based on current workload requirements, and being able to partitioning available physical resources, leads to more efficient utilization in the cluster.

VM hypervisors scale resources through device virtualization. Software-emulated devices appear to the VM guest as an I/O device, but all functionality is handled in the VM implementation. Paravirtualized devices also offer device functionality in software, but the software-defined device is backed by hardware and often resemble the physical device closely. As both methods of resource

✉ Jonas Markussen
jonassm@dolphinics.com

[1] Dolphin Interconnect Solutions, Oslo, Norway
[2] Simula Research Laboratory, Oslo, Norway
[3] University of Oslo, Oslo, Norway
[4] Simula Metropolitan Center for Digital Engineering, Oslo, Norway

virtualization require facilitation in the hypervisor, the availability of different types of resources is limited by the underlying virtualization technology being used. Furthermore, workloads that rely on multi-device interoperability become a challenge, as setting up necessary memory mappings for Remote Direct Memory Access (RDMA) and direct access between devices is generally not possible without extensive facilitation in both the hypervisor and interconnection technology. In many cases, RDMA functionality for paravirtualized devices even requires support in the VM guest drivers themselves.

In this context, a processor's I/O Memory Management Unit (IOMMU) enables devices to be *passed through* to a VM instance. A hypervisor can facilitate direct access to hardware 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 physical resources they use; distributing VMs across hosts in a cluster in a way that maximizes utilization becomes a challenge.

For clusters of machines interconnected with PCI Express (PCIe), we propose a different strategy to efficient resource sharing called Device Lending [1, 2]. In these clusters, I/O devices and interconnection technology are attached to the same PCIe fabric. Device Lending exploits the memory addressing capabilities inherent in PCIe in order to decouple devices from the hosts they physically reside in, without requiring any application- or device-specific distribution mechanisms. This decoupling allows a remote resource to be used by any machine in the cluster as if it is locally installed, without requiring any modifications to device drivers or application software. However, our previous implementation lacked support for dynamically discovering the guest physical memory layout. Because of this, it was necessary to limit the VM guest's available memory in order to force certain addresses used for device memory.

In this paper, we have extended our Linux Kernel-based virtual machine (KVM) support from [2] with a mechanism for probing the memory used by the VM guest in order to dynamically detect the guest physical memory layout. This makes it possible to map device memory regions for other pass-through devices, without requiring any manual configuration of the VM instance. Such devices can then access each other, using PCIe peer-to-peer transactions. With this kind of virtualization support, it is possible to enable custom configurations of multiple devices that are passed through to VMs and enabling fast data transfers between them. In addition, we have also implemented full interrupt support, something that was missing in our previous implementation.

We present our experimental performance evaluations of multi-device configurations using GPUs and enabling peer-to-peer between them, and compare our results to bare-metal experiments. Our findings depict 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 and the hardware address translation. We also evaluate the performance impact of increasing the distance between devices and CPUs, particularly focusing on the impact of I/O address virtualization. Finally, we present the applicability of using the system for a realistic I/O-intensive workload, i.e., running medical image classification via deep neural networks using remote GPUs and a remote NVMe drive. We can observe that the system makes bare-metal remote execution as efficient as local execution. Our results demonstrate that Device Lending offers a highly flexible I/O infrastructure in a PCIe cluster for both VMs and bare-metal machines, allowing dynamic compositions of local and remote I/O devices.

The remainder of this paper is organized as follows: we present essential capabilities of PCIe in Sect. 2. In Sect. 3, we discuss related work. In Sect. 4, we provide an outline of our original Device Lending implementation. We describe how we have extended Device Lending with virtualization support in Sect. 5. Section 6 describes how we have added support for borrowing devices from multiple lenders. We present our performance evaluation in Sect. 7, followed by a discussion of our findings and potential improvements in Sect. 8. Finally, we conclude the paper in Sect. 9.

2 PCIe overview

PCIe is today the most widely adopted industry standard for connecting hardware peripherals (devices) to a computer system [3]. Device memory, such as register and onboard memory is mapped into an address space shared with system memory (Fig. 1). Memory operations, such as reads and writes, are transparently routed onto the PCIe fabric, enabling 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 to 16 lanes. Each lane is a full-duplex serial connection, data is striped across multiple lanes, and broader links yield higher bandwidth. The current revision, PCIe Gen3 [4], has a throughput of around 13 GB/s for a x16 link.

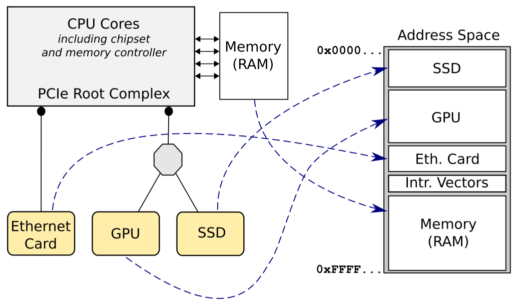


Fig. 1 Device memory is mapped into the same address space as the CPUs, allowing devices to access both system memory and other devices

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, ensuring 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 Fig. 2, the entire PCIe network is structured as a tree, where devices form the leaf nodes. In PCIe terminology, 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 (CPU cores, chipset, and memory controller). The entire PCIe network comprises the "fabric".

Some PCIe devices may support multiple functions, which appear to the system as a group of distinct devices, each which a separate set of resources. The term "device" actually refers to an individual function. An example of a multi-function device is a multi-port Ethernet adapter, where individual ports can be implemented as a separate functions.

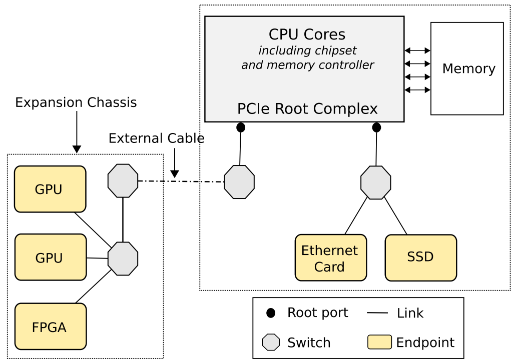


Fig. 2 Example of a PCIe topology using an external transparent link. The devices in an expansion chassis are attached to the same PCIe root as the internal devices, and are mapped into the same address space by the system

2.1 Memory addressing and forwarding

The defining feature of PCIe is that device memory and registers are mapped into the same address space as system memory (Fig. 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 I/O 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. In Fig. 2, the internal switch in the expansion chassis is connected to the root through an external transparent link (which differs from non-transparent links). The internal switch will have the combined downstream address range of both GPUs and the FPGA, allowing TLPs to be routed directly between them without passing through the root. This is referred to as peer-to-peer in PCIe terminology.

Another significant feature of PCIe, is the use of message-signaled 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. MSI-X is an extension to MSI with support for more than one address, allowing up to 2048 different, targeting specific CPUs and mandatory 64-bit addressing support.

2.2 Virtualization support and pass-through

Modern processor architectures implement IOMMUs, such as Intel VT-d [5]. The IOMMU provides a hardware virtualization layer between I/O devices and the rest of the system, including main memory. The defining feature of the IOMMU is the ability to remap addresses of DMA operations issued by any I/O device [6]. In other words, it translates virtual I/O 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 [7], or SR-IOV [8]. 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 [9]. This can be accomplished using an NTB implementation [10]. 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 [11].

Despite the name, an NTB actually appears as a PCIe endpoint. This is illustrated in Fig. 3, 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 separate address spaces, 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.

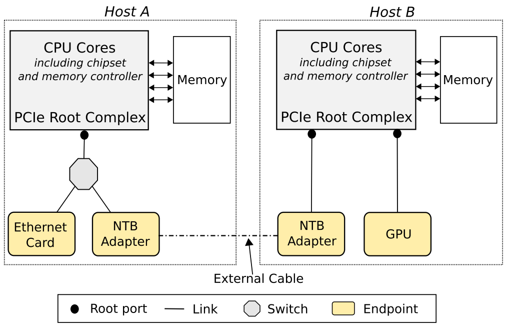


Fig. 3 Two independent networks are connected together using an NTB. The NTB Translates I/O addresses between the two different address spaces, creating a shared address space between the networks

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 [12] and SCI [13]. However, these ideas never got implemented, because none of these technologies were picked up for internal I/O interconnection networks.

PCIe is the dominant standard for internal I/O 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 I/O devices between multiple hosts.

3.1 Distributed I/O 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 [14, 15]. To make use of their high throughput, they rely on RDMA [16]. Variants are summarized by Huang et al. [17] 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 [18] and rCUDA [19] have been developed. Those middleware extensions have also been extended with device-specific protocols like GPUDirect for RDMA [20, 21] or NVMe over Fabrics.

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

While RDMA extensions may achieve very high throughput on the interconnection links, they are not as closely integrated with the I/O 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. This is in contrast to Device Lending, which works for all PCIe devices and does not require any changes to drivers.

A proposed approach for overcoming the protocol translation overhead would be to integrate network interface functionality directly into SoCs [22], but the improvement only takes effect when the SoCs are in communication with each other. This idea is followed in the rack-scale architecture [23], which generalizes a trend returning from switched cluster architectures to hypercube architectures [24, 25]. 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 I/O Virtualization (MR-IOV) [26] specifies how several hosts can be connected to the same PCIe fabric. The fabric is logically partitioned into separate virtual PCIe network trees, 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. Devices that are not multi-root aware can only be part of one PCIe root at the time. Despite being standardized in 2008 [26], 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 [27]. However, this approach only works for SR-IOV devices, while Device Lending makes no distinction between SR-IOV virtual devices and physical devices.

An additional virtualization approach is the Ladon system [28]. Ladon uses all PCIe and virtualization features as proposed in this paper, and is also implemented using NTBs. However, it achieves less freedom than our Device Lending, as devices are installed in a dedicated management host that manages the devices and distributes them to different remote guest VMs. In addition, devices can only be shared between different remote guest VMs, while Device Lending supports both VMs and bare-metal machines using the devices. In order to avoid management hosts becoming single points of failure, Ladon has been extended with fail-over mechanisms between management hosts in a master-slave configuration [29]. Device Lending is fully decentralized and thus avoids this all together.

Microsemi PAX [30] uses specialized PCIe switches that allow virtualization. The downstream switch ports reserve a large address range, called “synthetic endpoints”, which is similar to memory reserved by an NTB. Devices can then be hot-added through the virtual switch ports by remapping the synthetic endpoints to an actual device.

3.3 Partitioning the fabric

Rack-scale computers are so-called converged infrastructure systems, where both I/O 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 I/O devices requires additional modules, often only available from the same vendor.

Last but not least it should be mentioned that there have been some efforts in achieving live-partitioning using PLX PCIe switches [31], but a performance evaluation of this appears to be lacking.

4 Device lending

As illustrated in Fig. 4, 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 [1] for an unmodified Linux kernel using Dolphin’s NTB hardware and driver. 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 [9] 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

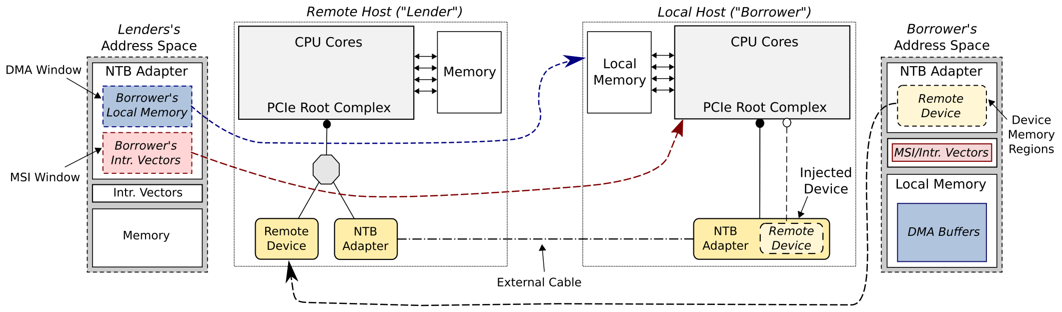


Fig. 4 Using an NTB, it is possible to map the memory regions of a remote device so local CPUs are able to read and write to device registers. The remote system can in turn reverse-map the local system's memory and CPUs for the device, making DMA and MSI possible. Device Lending injects a hot-added device into the Linux kernel device tree using these mappings

registers without being aware that the device is actually remote.

The lender is responsible for setting up reverse mappings for DMA and MSI.[2] As mentioned in Sect. 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 I/O 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, allowing devices to be temporarily assigned and reassigned to different systems. As hosts can act as both lender and borrower, we have created a highly flexible method of sharing devices (Fig. 5). This has advantages over distributed I/O using traditional approaches; network interfaces can be assigned to a computer while it needs high throughput, and released when it is no longer needed; access latency in NVMe over Fabrics using RDMA can be eliminated by borrowing the NVMe disk instead and accessing it directly, as shown in Fig. 6; large-scale CUDA programming tasks can make use of multiple GPUs that appear to be local instead of relying on middleware such as rCUDA [19]. In contrast to RDMA solutions, Device Lending works for all standard PCIe devices, and does not require any additional support in drivers.

Our original implementation, as described in [1], did not account for peer-to-peer 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 I/O addresses to other borrowed devices, in order to fully achieve device-to-device data transfers. In addition, our original implementation lacked support for borrowers that are VM guests. Adding virtualization support greatly increases 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 machine borrowers

Many modern architectures now implement IOMMUs, allowing DMA and interrupts to be remapped. This makes it possible for a hypervisor to grant access a driver running in a VM access to a physical device directly, without breaking out of the memory isolation, by using I/O virtual addresses. In Linux, such pass-through of devices is supported in the KVM hypervisor using the Virtual Function I/O API (VFIO) [32]. 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 hypothetical 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. However, this

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

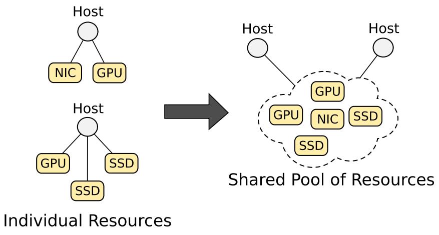


Fig. 5 Device Lending decouples I/O resources from physical hosts by allowing devices to be reassigned to hosts that currently need them. We imagine this as hosts in the cluster contributing to a shared pool of I/O resources that can be cooperatively time-shared among them

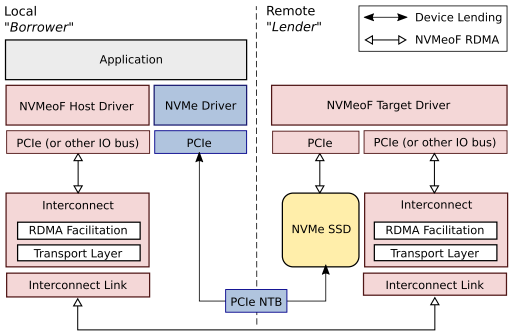


Fig. 6 Illustration of native NVMe using Device Lending compared to NVMe over Fabrics using RDMA. Device Lending makes remote devices appear as if they are locally installed and there is no need for specialized support in devices or drivers

approach would not be feasible due to the following reasons:

- 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 leads to poor utilization of device resources.
- All devices borrowed by the same physical host would be placed in to the same IOMMU domain by Device Lending. VFIO requires that pass-through devices must be be placed in a per-guest IOMMU domain managed by VFIO. This is required in order to prevent memory accesses that could potentially break out of the memory isolation provided by virtualization.
- VFIO requires the entire address space of the VM to be mapped for the device. As there is no method of knowing which physical memory pages will be allocated for the VM instance before it is running, establishing this mapping in advance would require mapping all physical memory. We instead need a mechanism for only pinning and mapping the memory pages used by the VM instance in order to create necessary DMA windows.

In the 4.10 version of the Linux kernel, an extension to VFIO called mediated devices [33] 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. When a VM guest accesses the virtual device, certain operations, such as accesses to the device's configuration space or setting up interrupts, are intercepted by the mediated device parent driver. The idea is that a single physical device can be used to emulate multiple virtual devices, while still allowing some direct access to hardware. In our case, using the mediated devices extension provides us with finer grained control over what the hypervisor and guest OS is attempting to do with the device than with "plain" VFIO.

Our implementation registers an mediated device parent device for devices used by Device Lending without borrowing them first. This allows KVM to pass through the device to a VM guest without it being borrowed (and locally injected) first. Only when the guest OS boots up and resets the device, do we actually borrow the device and take exclusive control. When the guest OS releases the device, either by shutting down or because the device is hot-removed, we return the device. Not only does this limit the lifetime of a borrowed device to only when the VM is running and using the device, but it also makes it possible to hot-add a device to a live VM instance if the VM emulator supports it.

As we now have control over when a device is being used and which VM instance is using it, we can set up the appropriate isolated IOMMU groups on the lender. As shown in Fig. 7, this allows a device to be mapped in to the same virtual address space (guest-physical) as the VM as well as providing the necessary isolation to protect against rogue memory accesses. We also set up IOMMU mappings on the local system, in order to map continuous memory ranges to physically scattered memory on the host over the NTB.

While other implementations using mediated devices 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 just 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. We are also not able

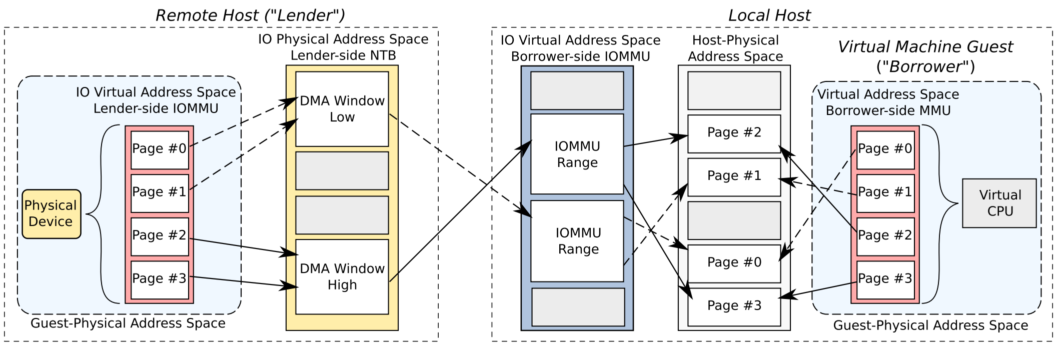


Fig. 7 By using IOMMUs on both sides of the NTB, it is possible to map a physically remote device into a local VM guest's address space. The borrower-side IOMMU provides continuous memory ranges that can be mapped over the NTB, while the lender-side IOMMU allows the device to be mapped into an address space using the same guest-physical addresses used by the VM

to know in advance what memory pages will be used 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. In addition, the mediated device API does not provide any information about the guest-physical memory layout, which we need to know which address ranges to map for the device.

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 intercepts when a configuration cycle enables this register, and only then notifies KVM to pin the necessary memory pages. With the pages now locked in memory, we are able to properly set up DMA windows to memory used by the VM instance. The x86 architecture uses well-defined addresses for low and high memory. We are able to discover how much memory the VM has allocated by attempting to pin memory starting at these addresses. In this way, we are able to dynamically detect the guest-physical memory layout.

Finally, VFIO and mediated devices use the *eventfd* API to trigger interrupts in the VM instance. Our current implementation 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-side request handler is invoked. This handler must then 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 of our solution 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. We have also improved Device Lending in general with support for 64-bit MSI/MSI-X.

6 Supporting multiple devices and peer-to-peer

Some processing workloads may require the use of multiple I/O devices and/or compute accelerators, in addition to moving data between them in an efficient manner. This often involves the use device-to-device DMA, as described in Sect. 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 shortest-route peer-to-peer transactions directly between devices in the fabric is not possible when using an IOMMU, and TLPs must traverse the root (Fig. 8). PCI-SIG has developed an extension to the transaction layer protocol that allows devices that have an understanding of I/O virtual addresses to cache resolved addresses [34], 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 efficiently, the

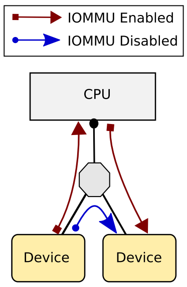


Fig. 8 IOMMUs introduce a virtual I/O address space for devices. Peer-to-peer transactions between devices is routed through the root in order for the IOMMU to resolve virtual addresses to physical addresses

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 I/O 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 I/O 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 I/O 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 I/O addresses to the memory regions of *target*. If IOMMU is disabled, then it is simply a matter of returning the local I/O addresses of 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 I/O 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.

An additional consideration is required if the borrowing machine is a VM. In this case, *target* is already mapped into the guest-physical address space of the VM guest. The memory regions of *target* must be mapped for *source* using these exact addresses. Since the VM case already uses the lender-side IOMMU, as explained in Sect. 5, we can simply use the IOMMU of *source*'s lender and specify the addresses that correspond to the VM guest's view of the address space.

7 Performance evaluation

In order to evaluate our improved Device Lending implementation, we have done extensive evaluations of the bandwidth and latency of peer-to-peer DMA transfers. As VM pass-through require the use of an IOMMU on the lending system, we particularly focus on the impact I/O address virtualization has on performance with regards to longer data paths. For all our comparisons, we present the topology and machine configurations and compare performance for native bare-metal borrowers and VM borrowers. Our baseline comparison for all evaluations are running locally, on a bare-metal machine.

In Sect. 7.5, we prove the capability of running unmodified software and device drivers by presenting the performance of an unmodified convolutional neural network-based application, using the Keras framework with Tensorflow. We argue that running unmodified code using a complex machine learning framework on commodity hardware demonstrates the strength and flexibility of our Device Lending approach.

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 I/O addresses (Fig. 8). 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 fabric 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 Fig. 9. Two Intel Xeon machines are connected together with Dolphin's PXH830 NTB host adapters [35] 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* [36] 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 10 transfers and then report the mean bandwidth. We have repeated this 10 times.

Figure 10 shows the reported mean bandwidth for both DMA writes and DMA reads, comparing the performance of shortest path (Rem-SW) with TLPs being routed through the root (Rem-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 emitted out again as 4×64 byte read requests on the other side of the root. As read performance is already limited by the number of requests they are able to keep open, requesting less data at a time 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 also compared our results to running locally on the lender without using Device Lending (Loc). The achieved bandwidth of the local run is slightly better than our peer-to-peer performance for smaller transfer sizes; this is most likely due to the fact that the GPU is further 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

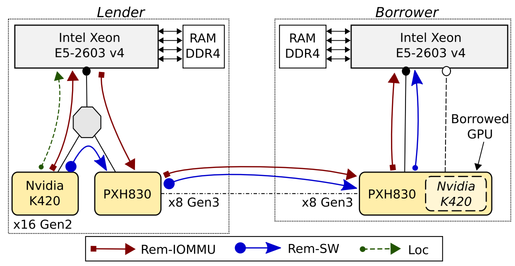


Fig. 9 Configuration used in our IOMMU evaluation. The borrower is using the remote GPU. When the lender-side IOMMU is enabled, TLPs are routed through the lender's root before going over the NTB (Rem-IOMMU). We have also compared to a baseline comparison, running locally on the lender machine itself (Loc)

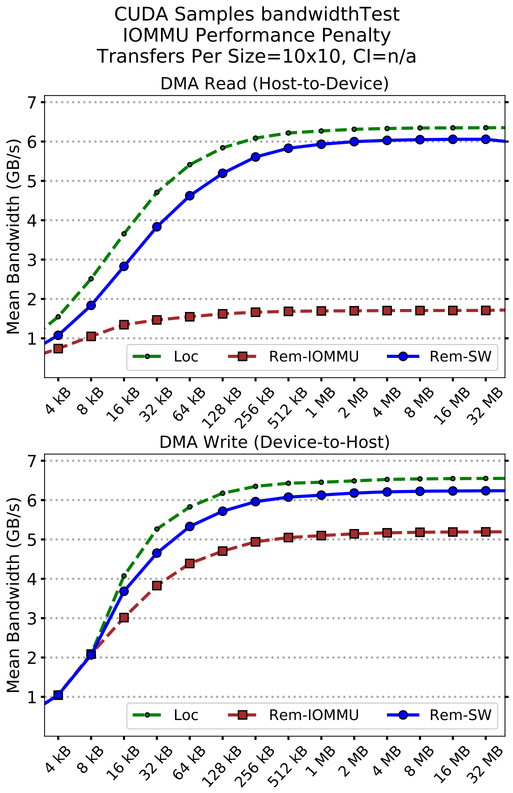


Fig. 10 Reported bandwidth for different transfer sizes using an unmodified version of the *bandwidthTest* CUDA samples program

significance of this additional latency decreases and the reported bandwidths starts to converge.

7.2 Native peer-to-peer evaluation

In order to evaluate our multi-device support, we have measured the performance of peer-to-peer DMA transfers between two Nvidia Quadro K420 GPUs. The machines are connected together using the PXH830 NTB adapters in a three-way configuration, providing a separate Gen3 x8 link between all three machines. The K420 GPUs are Gen2 x16, which provides roughly the same bandwidth as Gen3 x8.

Figure 11 shows the three different hardware configurations used in this evaluation:

- A local machine using two GPUs installed in the same local host, illustrated in Fig. 11a. This is our baseline for comparing the performance of using remote devices vs. local devices. Since it is not possible to enable peer-to-peer transfers on a local machine using the IOMMU, we instead force transfers to be routed through the root by placing the GPUs behind different PCIe switches.
- A local machine (borrower) using two remote GPUs, installed in a single remote host (one lender). This is illustrated in Fig. 11b.
- A local machine (borrower) using two remote GPUs, installed in different remote hosts (two lenders). This is illustrated in Fig. 11c.

A complete list of the scenarios are given in Table 1. Note that in Fig. 11, we have only highlighted the data path for peer-to-peer DMA writes with the IOMMU enabled and disabled. We compare the performance benefit of direct device-to-device DMA writes, using peer-to-peer transactions, to transfers via RAM, where one GPU first writes to RAM and the other reads from it using DMA. In order to do this, we have developed two CUDA [36] applications for measuring transfers from one GPU to another. Note that we do not use any special semantics or other userspace software to make this CUDA program work for borrowed remote GPUs, using Device Lending they simply appear to the CUDA programs as if they are locally installed.

7.2.1 Bare-metal bandwidth evaluation

The first of the two CUDA programs measures the bandwidth of DMA transfers from one GPU to another using two different transfer "modes". The first mode is enabling peer-to-peer transactions, allowing one GPU to write directly into another GPUs memory. The second mode is hosting an intermediate buffer in system memory (RAM), where one GPU first writes to that buffer, followed by the other GPU reading from it afterwards. We record a CUDA event before and after each scheduled transfer, and we also schedule a dummy CUDA kernel launch in order to prevent our bandwidth measurements being affected by the CUDA driver's ability to pipeline transfers.[3]

Figure 12 shows the bandwidth for all three configurations (depicted in Fig. 11). We have recorded the completion time for 1000 individual DMA transfers of a given size, for each transfer size shown along the X-axis, and plot the mean bandwidth. We also show the 95% confidence interval as a filled-out area around the respective lines. The top row shows our peer-to-peer transfers, while the bottom row shows transfers via system memory. We also show the difference in performance when the IOMMU is enabled and disabled on the lender(s), where the GPUs reside. Note that in our local comparison, we place the GPU behind a different PCIe switch in order to force TLPs to traverse the root, since it is not possible to enable the IOMMU in this scenario.

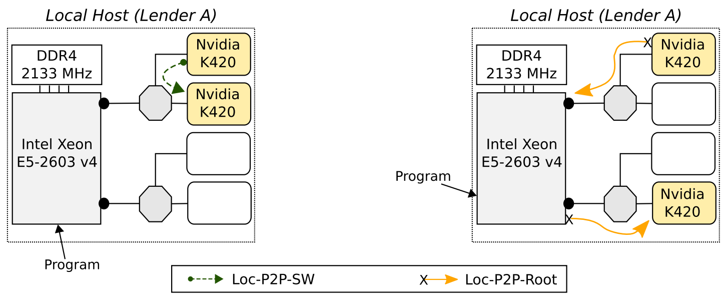
Using peer-to-peer DMA writes (top row), we see that the achieved bandwidth is almost the same as our local comparison in the same lender scenario: 1L-P2P-SW is almost identical to Loc-P2P-SW, and 1L-P2P-IOMMU is almost identical to Loc-P2P-Root. Even though the GPUs are remote, the data path between the GPUs are similar. For smaller transfer sizes, the local transfers achieve slightly higher bandwidth. However, when the transfer size increases, the lines converge, and for transfers of 4 megabyte and above, the difference becomes negligible. We suspect that the protocol used by the driver in order to synchronize the GPU and schedule DMA transfers may involve some reading and writing over the NTB. For small-sized transfers, this additional latency relative to the transfer size has an effect.

When the GPUs reside in different lenders, the data path is increased, which has an expected impact on performance. As shown in the 2L-P2P plot (top row, to the right) in Fig. 12, particularly when the IOMMU is enabled, the increased number of hops impacts the performance.

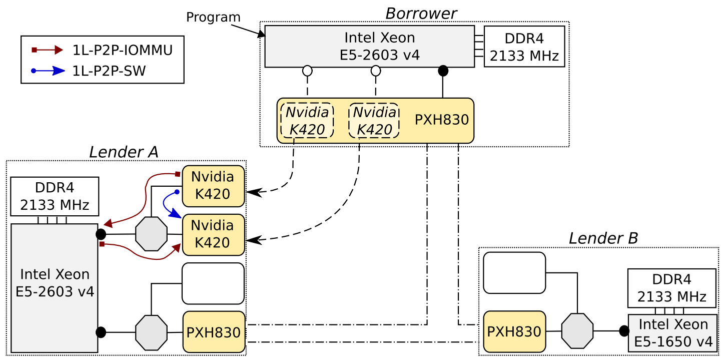
The second mode of our program was used to evaluate "bouncing" via system memory. By hosting a memory buffer in RAM, one GPU has to first write to this buffer before the other GPU in turn can read from it. Borrowing remote GPUs using Device Lending, the distance between system memory and GPU is now increased, and the impact of this is visible, as illustrated in Fig. 12 (bottom row). We see that transfers that do not cross the root (2L-RAM-SW)

[3] The CUDA samples *bandwidthTest* program, used in Sect. 7.1, schedules 10 rapid copy operations at the time and reports the average of these ten, allowing the CUDA driver to pipeline transfers and optimize small transfers.

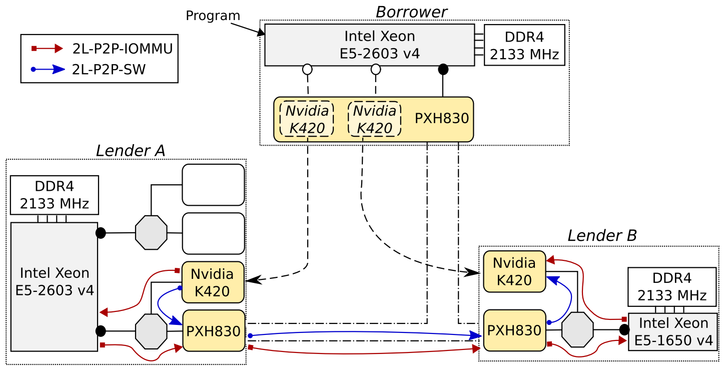
Fig. 11 The three-node cluster configurations used in our bare-metal multi-device evaluation, showing the the data paths for direct peer-to-peer write transactions



(a) Two GPUs used by a local instance. We force transactions to traverse the root in the Loc-P2P-Root scenario in lieu of an IOMMU, shown on the right-hand side.



(b) Two GPUs borrowed natively from the same lender. Note how the data paths are similar to the local scenarios in Figure 11a.



(c) Two GPUs borrowed natively from two different lenders.

are very similar to our baseline local comparison (Loc-RAM-Root). However, similarly to what we observed in Sect. 7.1, DMA reads are significantly affected by TLPs traversing the root, as this drastically reduces the link utilization. This is seen in 1L-RAM-IOMMU and 2L-RAM-IOMMU, where the reported bandwidth drops drastically.

Table 1 The different scenarios used in our bare-metal peer-to-peer evaluation. Note the number of hops and CPU roots transfers have to traverse

Name	Scenario	Transfer	Roots	Hops
Loc-P2P-SW	Two local GPUs installed in same machine as driver.	Peer-to-peer	0	1
Loc-P2P-Root	Two local GPUs installed in same machine as driver.	Peer-to-peer	1	3
Loc-RAM-Root	Two local GPUs installed in same machine as driver.	Via local RAM	1	3
1L-P2P-SW	Two remote GPUs borrowed from the same lender.	Peer-to-peer	0	1
1L-P2P-IOMMU	Two remote GPUs borrowed from the same lender.	Peer-to-peer	1	3
2L-P2P-SW	Two remote GPUs borrowed from different lenders.	Peer-to-peer	0	4
2L-P2P-IOMMU	Two remote GPUs borrowed from different lenders.	Peer-to-peer	2	8
1L-RAM-SW	Two remote GPUs borrowed from the same lender.	Via borrower's RAM	3	11
1L-RAM-IOMMU	Two remote GPUs borrowed from the same lender.	Via borrower's RAM	3	11
2L-RAM-SW	Two remote GPUs borrowed from different lenders.	Via borrower's RAM	1	8
2L-RAM-IOMMU	Two remote GPUs borrowed from different lenders.	Via borrower's RAM	3	11

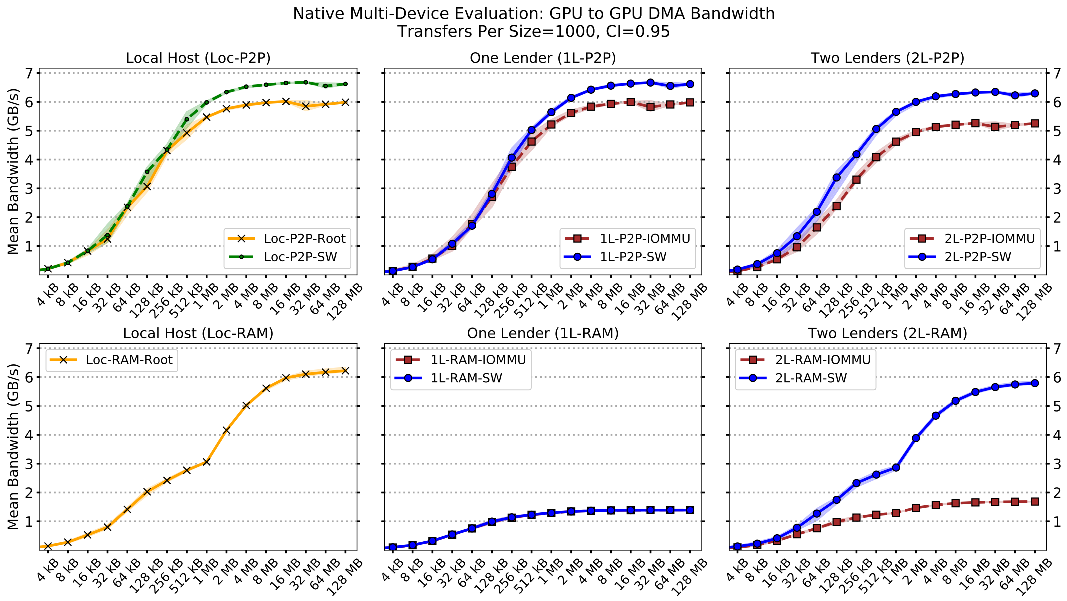


Fig. 12 Mean DMA bandwidth for different transfer sizes. The filled-out area represents the 95% confidence interval. The top row shows writes using peer-to-peer, while the bottom row shows "bouncing" via RAM. For the peer-to-peer, we achieve almost the same bandwidth as our local comparison. For transfers via RAM, the bandwidth is reduced by read TLPs traversing through CPU roots

It is interesting to note that 1L-RAM-IOMMU and 1L-RAM-SW both traverse the root, but the IOMMU is respectively on and off. This strengthens our suspicion that the issue is TLPs being routed through the root, and not necessarily some effect of using the IOMMU alone.

A simplified illustration of the data path for the full list of scenarios is shown in Fig. 13. Note that each additional "hop" in the path adds additional latency to the TLP completion time, something that particularly affects reads.

Our peer-to-peer bandwidth evaluation indicates that it is possible to achieve close to local performance. For DMA write operations, the performance of a local program using borrowed remote devices is comparable to using local devices. Note that while DMA reads are affected by the increased distance between the device and the memory it reads from, it is expected for longer data paths and not an effect of our Device Lending mechanism.

7.2.2 Bare-metal latency evaluation

Using CUDA, there are two ways of initiating DMA transfers; either the CPU can initiate DMA transfers, or the device can do it itself. The first approach is similar to the CUDA samples program *bandwidthTest*. The second approach is possible using CUDA's unified memory model, where a CUDA kernel can access system RAM directly through a memory pointer. This eliminates the need for an explicit copy to GPU memory operation. With unified memory, it is also possible for one GPU to directly access memory of another GPU, using peer-to-peer DMA.

As shown in Sect. 7.1, we suspect that the increased distance between CPU and device affects the time it takes for the driver to synchronize with the device and initiate a transfer. We also observed a similar effect for smaller-sized transfers in Fig. 12. Therefore, we developed a second CUDA program in order to measure peer-to-peer latency more accurately. Using CUDA kernels and allowing the GPUs themselves to initiate transfers, we eliminate any synchronization overhead caused by the driver (running on the local CPU). One GPU is tasked with increasing a counter, writing it to the other GPU's memory pointer and waiting for an acknowledgement before continuing (*ping*). The other GPU waits for the counter to increase by one, and acknowledges by writing back to the first GPU's memory pointer (*pong*). The whole roundtrip is measured by recording the current GPU clock cycle count and divide it by the clock frequency, giving us the full *ping–pong* latency.

The memory used for our counter can either be hosted in GPU memory or in RAM. The difference is that in the peer-to-peer scenarios we eliminate any DMA read operations and the GPUs are able to write directly to GPU memory. When memory is hosted in RAM, 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 Fig. 13. Note that each additional "hop" in the total path adds additional latency to the overall completion time.

Figure 14 shows the mean ping-pong latency for all scenarios. We measured the latency for 100,000 ping-pongs, and the error bar depicts the 99% confidence interval. For comparison, the one-way RAM-to-RAM memory latency between the borrower and Lender B was measured to around 700 nanoseconds, where the NTB itself adds 350-365 nanoseconds. When GPUs reside behind the same switch (1L-P2P-SW), we achieve the same latency as our local comparison (Loc-P2P-SW). We also see the same when the IOMMU is enabled (1L-P2P-IOMMU) and the local comparison (Loc-P2P-Root). Again, this demonstrate that our Device Lending mechanism does not add any overhead.

We also observe that the latency increases according to the increased data paths (illustrated in Fig. 13), as expected. The latency for 2L-P2P-SW increase with a little more than 700 nanoseconds, compared to 1L-P2P-SW (and Loc-P2P-SW), which corresponds with the 350 nanoseconds added by the NTB (in one direction). In the scenarios where the counter memory is hosted in the borrower's RAM, the latency increases significantly because both GPUs now have to read in addition to writing. Our latency evaluation show that the latency of reading and writing is only affected by the path, and achieving the same latency as our local comparison when the path is similar.

7.3 VM peer-to-peer evaluation

We have also evaluated peer-to-peer performance for devices passed-through to a VM. We installed Ubuntu 16.04 with CUDA 9.0 on an Intel P4800X NVMe drive. As our VM emulator, we used QEMU 2.10.1. Two Nvidia

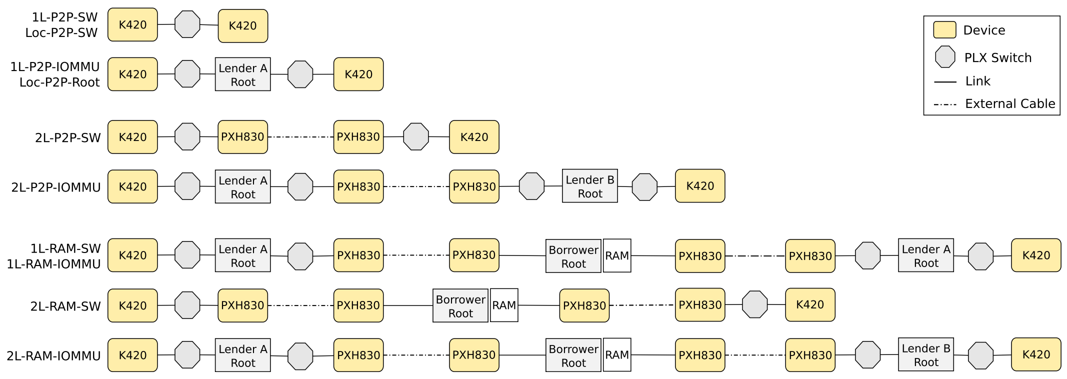


Fig. 13 Data paths for the different bare-metal scenarios. Each hop slightly increases the completion latency

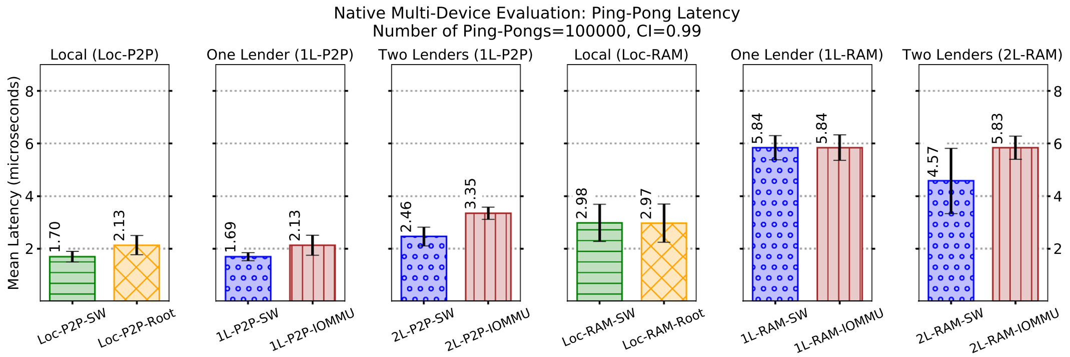


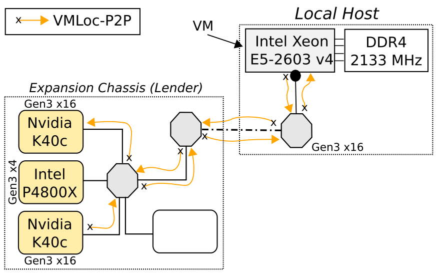
Fig. 14 Mean round-trip latency for 100,000 ping-pongs. The error bar represents the 99% confidence interval. For the peer-to-peer scenarios, we achieve the expected latency corresponding to the data path. When bouncing through RAM, the latency increases drastically due to the second GPU having to read from RAM

Tesla K40c GPUs along with the boot disk was passed through to a local VM using standard VFIO pass-through [32] with KVM, and to a remote VM using our Device Lending implementation. We used the same two CUDA programs from Sect. 7.2 for measuring bandwidth and latency respectively.

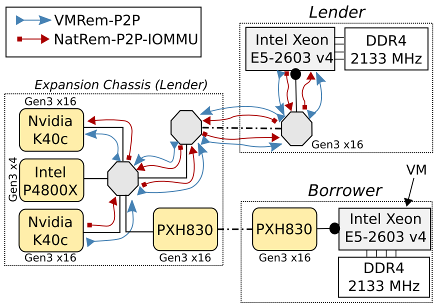
Figure 15 depicts the topologies used for these tests. The GPUs, the disk and the NTB adapter are located in an expansion chassis connected with a transparent link to the lender. TLPs must be routed through the lender's root before they can be transmitted over the NTB (which also resides in the expansion chassis), making this this configuration suboptimal for running a remote VM. As such, it serves as a worst-case scenario for running a VM, especially for the scenario where transfers are bounced via RAM. Figure 17 shows the data path for all scenarios. We have also included a native remote comparison using Device Lending where the IOMMU is enabled to illustrate any virtualization overhead. Note that the data path is similar for both local and remote scenarios when the devices use peer-to-peer DMA. The evaluated scenarios are listed in Table 2.

7.3.1 VM bandwidth evaluation

Figure 16 depicts the measured bandwidth for all configurations, using the same CUDA program as in Sect. 7.2. For each transfer size, we plot the mean reported bandwidth of 1000 transfers. We also show the 95% confidence interval as a filled-out area surrounding the plotted lines. The upper-most plot depicts direct peer-to-peer transfers between the GPUs. We compare our Device Lending mdev implementation, with two borrowed remote GPUs passed to a local VM (VMRem-P2P), to a local comparison, or baseline, where two local GPUs are passed to a local VM



(a) Local VM instance using VFIO pass-through.



(b) Remote VM borrower and remote native borrower. Note the data path is similar when the lender-side IOMMU is enabled.

Fig. 15 Topologies used in our VM peer-to-peer evaluation. We have compared a local VM using VFIO pass-through to a remote VM using our extended Device Lending. Note that the devices are located in an expansion chassis, which increases the number of hops to the lender

(VMLoc-P2P). As with our previous bandwidth evaluations, we see a similar pattern as before: timing and synchronization between driver and GPUs appear to affect smaller-sized transfer, but becomes less relevant when the

Table 2 The different scenarios used in our VM peer-to-peer evaluation. Since the GPUs and the NTBs are now attached in an expansion chassis, the number of hops is very high when the IOMMU is enabled

Name	Scenario	Transfer	Roots	Hops
VMLoc-P2P	Two GPUs installed in same, local expansion chassis.	Peer-to-peer	1	7
VMRem-P2P	Two GPUs installed in same, remote expansion chassis.	Peer-to-peer	1	7
NatRem-P2P-IOMMU	Two GPUs installed in same, remote expansion chassis.	Peer-to-peer	1	7
VMLoc-RAM	Two GPUs installed in same, local expansion chassis.	Via local VM's RAM	1	7
VMRem-RAM	Two GPUs installed in same, remote expansion chassis.	Via borrowing VM's RAM	3	21
NatRem-RAM-IOMMU	Two GPUs installed in same, remote expansion chassis.	Via borrower's RAM	3	21

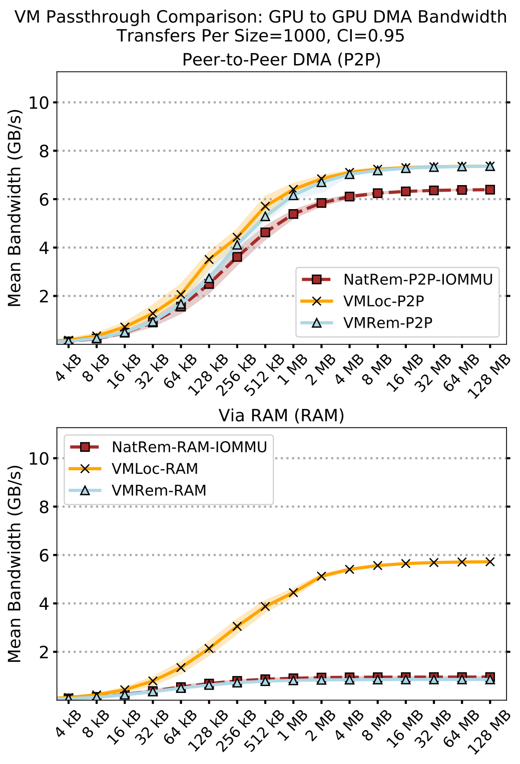


Fig. 16 Mean bandwidth for 1000 transfers per transfer size. 95% confidence interval. For peer-to-peer transactions we achieve the same bandwidth as running locally

transfer sizes increases. At around 4 megabytes this overhead is insignificant.

We have also included an additional comparison, namely a remote bare-metal machine borrowing the two remote GPUs and using them natively (NatRem-P2P-IOMMU). In order to force TLPs to traverse the same route as our KVM implementation (where lender-side IOMMU is required), the IOMMU is also enabled on the lender for the native comparison. It is interesting to note that it appears to achieve slightly lower bandwidth than when running in a VM, despite the data path being the same. We do not completely understand why this is the case.

The lower plot in Fig. 16 depicts transfers that are "bounced" via RAM. The memory buffer is allocated in system memory, and one GPU has to first write to it, before the other GPU can read from it. Since the lender's root is now even further away from the devices, read requests are significantly affected by the increased path. Combined with the reduced link utilization, as we observed in Sect. 7.1, the result is a drastic decrease in achieved bandwidth, for both our native Device Lending scenario (NatRem-RAM-IOMMU) and our KVM implementation (VMRem-RAM).

A simplified view of the data paths of all scenarios is illustrated in Fig. 17. We see that the path for NatRem-RAM-IOMMU and VMRem-RAM consists of 21 hops, traversing tree CPU roots, the NTB twice and the external transparent link four times. Note, however, that the performance for our VM implementation is similar to the native bare-metal performance, indicating that the Device Lending mechanism itself does not add any additional overhead. For the direct peer-to-peer DMA writes, the performance is comparable to the local comparison, which serves as our baseline.

7.3.2 VM latency evaluation

Using the second CUDA program, we also measured the ping-pong latency for the same scenarios. This is shown in Fig. 18, each bar is the mean reported latency for 100,000 ping-pongs (the error bar represents the 99% confidence interval). It is interesting to note that the latency for the remote scenarios using Device Lending (NatRem-P2P-IOMMU and VMRem-P2P) is actually slightly better than our local comparison, even though the data path is the same (Fig. 17). We assume this may be related to how VFIO exposes the GPU registers to the driver in the local case. In our KVM implementation, we expose the device memory regions directly, allowing the driver running in the VM guest to access GPU registers directly.

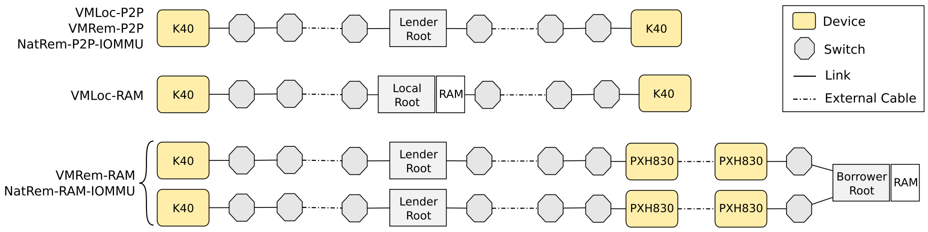


Fig. 17 Data paths for the different VM scenarios. Each hop slightly increases the completion latency. Because the NTB adapter is in the expansion chassis next to the GPUs, the number of hops when the lender-side IOMMU is enabled is very high

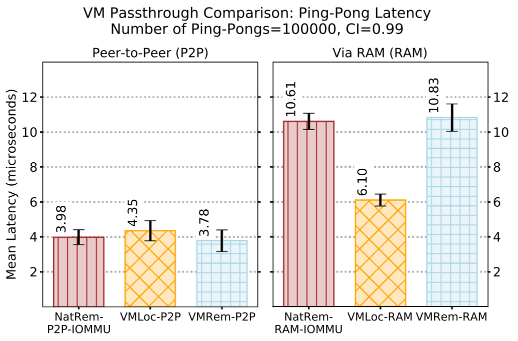


Fig. 18 Mean round-trip latency for 100,000 ping-pongs. 99% confidence interval. For peer-to-peer transactions, we achieve the same latency as running locally

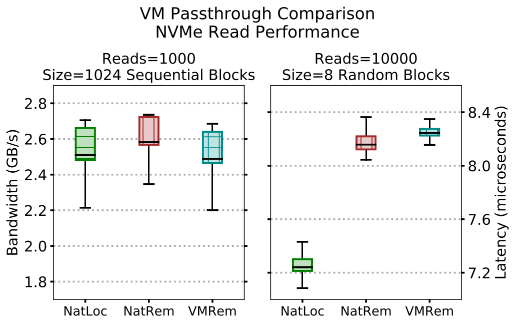


Fig. 19 Bandwidth and latency when reading from disk (DMA write). We read 1024 sequential blocks for measuring bandwidth, and 8 blocks with a random offset for latency

The increased latency of reading from remote RAM corresponds with the increased number of hops. The data path of running the VM locally (VMLoc-RAM) has only 7 hops, while the data paths of our remote native comparison (NatRem-RAM-IOMMU) and the remote VM (VMRem-RAM) both have 21 hops.

Our VM peer-to-peer evaluation indicate that we are able to achieve the same performance as a local VM when the data path is the same, and that we achieve the same performance as running natively (with lender-side IOMMU) even in the worst-case scenario.

7.4 Pass-through NVMe experiments

We have also performed experiments with our VM implementation using an Intel Optane 900P NVMe disk. We have compared the performance of the disk on a local machine without using Device Lending, a physical borrower (NatRem), and from a VM guest (VMRem). The machines are connected back-to-back using PXH830 NTB adapters [35]. The one-way 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 16.04 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. Device Lending requires Linux on the host.

Figure 19 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 (VMRem). Interestingly, we observe that the physical borrower (NatRem) achieves slightly higher median bandwidth than compared to the local baseline.

Latency was measured by reading 8 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 of 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 I/O commands, and then reach over the NTB again in order to post the I/O 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 (VMRem), only increases the latency slightly compared to the physical borrower (NatRem). Our evaluation show that it is possible to borrow a remote NVMe drive without any performance overhead beyond the added latency of the NTB. Additionally, it shows that our KVM extension to Device Lending is able to achieve almost the same bandwidth and latency as a native borrower.

7.5 Image classification workload

In order to demonstrate that Device Lending is applicable for real-world workloads, we run a GPU-intensive machine learning task. The program we use for the tests is a typical implementation of convolutional neural network (CNN) training in the Python machine learning framework Keras [37]. Keras is a higher level framework and wraps different lower level machine learning frameworks. In our case, Keras uses Tensorflow [38] as its backend. Keras allows multiple GPUs to work together by replicating the machine learning model being trained on each of the GPUs, then splitting the model's inputs into sub-batches which are distributed on the GPUs. When the GPUs are done, the sub-batches are concatenated on the CPU into one batch. This introduces quasi-linear speedup [39]. However, as our machine learning program can only run on a single system, we utilize multi-GPU support in Keras by borrowing remote GPUs and making them appear locally installed using Device Lending.

Our real-world workload is produced by a program that trains available models in Keras on given datasets with given hyperparameters using transfer learning [40]. Transfer learning is a technique for training datasets, where we use models with weights that are pre-trained on a dataset with similar classes to the dataset we want to train on. In our case, we use a VGG19 [41] model pre-trained on the ImageNet [42] dataset.

Transfer learning is done in two training steps: first, we remove the classification block of the pre-trained model, attach a new block corresponding to the number of classes in the dataset we want to train on, and train the new classification block only. In the second step, we train all the layers. For both steps, we use the stochastic gradient descent optimizer available in Keras. We ran the training on an 8-classes image dataset of the gastrointestinal tract called Kvasir [43–45].

We measured the runtime of a single epoch of the model training on two Nvidia K40c GPUs as well as loading images from storage and writing the results back using an Intel Optane P4800X. While a single epoch may not give the statistical significance needed for reliable machine learning results, we are only interested in the system performance. We used Keras 2.2.4 with a Tensorflow backend running on an Ubuntu 16.04 installation with CUDA 9.0 cuDNN 7.1. Both GPUs and the disk were used in all scenarios, and we also booted VMs and physical machines from the disk. For the native remote tests (NatRem-SW and NatRem-IOMMU) the disk was instead locally installed, in order to boot from it. The physical host had 16 GB memory and 6 CPU cores (Intel Xeon CPU E5-2603 v4). We reserved 4 cores and 8 GB for the VM, and used all 6 cores and the remaining 8 GB for the native run.

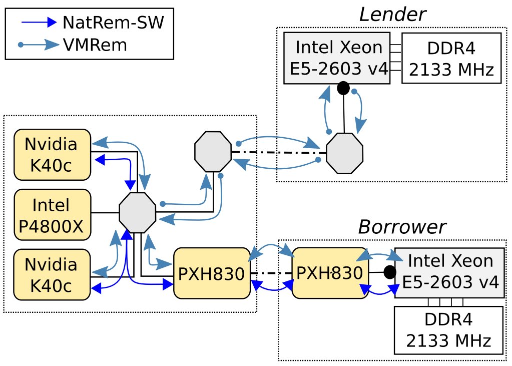


Fig. 20 Configuration used for our workload. The we have run a local native (NatLoc) and local VM (VMLoc) comparison on the lender and remote runs on the borrower (NatRem and VMRem). Note that this topology best-case scenario for the remote *native* peer-to-peer data path (NatRem-SW), while simultaneously being worst-case for remote VMs (VMRem)

Figure 20 depicts the topology of our evaluation. When using the multi-GPU model in Keras, the Tensorflow backend outputs a GPU peer-to-peer matrix, indicating that it is capable of direct DMA without bouncing via RAM.[4] We have used the same configuration for local and remote runs. As discussed in Sect. 7.3, this topology is a form of worst-case scenario for running remote VMs because the IOMMU address virtualization requires TLPs to be routed through the lender's root. At the same time, it is a best-case scenario for native runs with direct peer-to-peer transfers, as devices reside behind the same switch.

Figure 21 show the total runtime of the model training for the different scenarios. For the best-case scenario, running natively with direct data paths, we see that the remote run (NatRem-SW) runs as fast as the local native comparison (NatLoc). Enabling the IOMMU and forcing traffic through the lender's root (NatRem-IOMMU) increases the overall runtime. We have also compared a local VM using standard VFIO pass-through (VMLoc) to our KVM implementation (VMRem). It is interesting to note that the local VM runtime is higher than the remote native using the IOMMU. This indicates that virtualization adds some additional overhead compared to running on bare-metal. The VMs also use less memory and CPU cores than native. We suspect this is also the case for the remote

[4] We also confirmed that the GPU driver sets up peer-to-peer transfers by observing IOMMU mappings ranges.

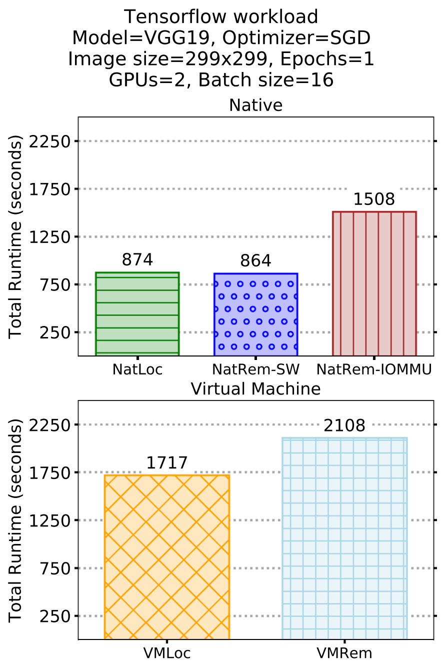


Fig. 21 Total runtime of Keras workload in different scenarios. There is a significant performance decrease when running in a VM and when the GPUs are remote. Note that the local VM (VMLoc) performs worse than the native remote, indicating that there is additional performance overhead caused by virtualization

VM (VMRem), which means that we get virtualization overhead in addition to the performance penalty of longer data paths through the IOMMU.

8 Discussion

Device Lending is a mechanism for decoupling devices from the hosts they physically reside in. Using hardware memory mappings, we facilitate the use of remote hardware resources without adding any software overhead [1, 2, 45]. We have extended our original Device Lending with KVM support for peer-to-peer transfers between multiple devices passed through to a VM. In this section, we discuss some considerations for borrowing devices from a VM guests.

8.1 I/O address virtualization

In our performance evaluation (Sect. 7), we observed that the data path in terms of number of hops affects the TLP completion latency. We also observed that using the lender-side IOMMU forces TLPs to be routed through the CPU on the lender. Our findings also seem to match previous performance evaluations of IOMMUs [46].

When the driver and the device frequently communicate with each other, as seen as synchronization overhead for small DMA transfers in our evaluations using Nvidia GPUs, it may affect performance since TLPs has to go back and forth over NTB. For larger DMA transfers, we observed that the significance of this delay decreases. For peer-to-peer transfers that do not require synchronization by the CPU, as is the case for our ping-pong evaluations, the distance between GPU and driver is insignificant. It should be noted that traversing the NTB adds less than half of the latency added by InfiniBand FDR adapters [15, 21]. For native peer-to-peer transfers with PCIe switches, where shortest-path routing is possible, we therefore argue that Device Lending can be used with extremely low performance overhead.

A major performance bottleneck occurs when DMA read requests are routed through the root, as the Intel Xeon CPUs used in our evaluations alter the read requests to request less data at the time (from 256 to 64 bytes). This leads to decreased utilization of the PCIe links. Since devices may be limited by the number of read requests they are able to keep open, the combination of poor link utilization and longer data paths can drastically affect the DMA bandwidth for some scenarios. However, we also observed a similar effect when read requests were routed through the CPU without the IOMMU being enabled. This strongly indicates that routing peer-to-peer through the CPU is a problem in general. Since the I/O address virtualization is required for *both* local and remote passthrough, it is worth investigating further by evaluating other CPU architectures that implement an IOMMU, such as AMD EPYC/Zen and IBM Power.

Our recommendation is to try to minimize the number of hops after the CPU in order to reduce the performance penalty of routing through the root, and to use shortest-path peer-to-peer transfers where possible. For bare-metal borrowers, this can be accomplished by disabling the IOMMU on the lender all together. For VMs, it may be possible to create a PCIe backplane that uses an NTB per device, allowing the NTBs to map the guest-physical address space for the devices rather than using an IOMMU for this.

Another possibility for avoiding IOMMU performance penalty, is using PCIe switches and devices that support caching of resolved virtual addresses using the standard for this specified by PCI-SIG [34]. Note that while it is also possible to disable the IOMMU on the borrower as well, this requires mapping the entire address space of borrower through the lender-side NTB and is therefore not practical with multiple borrowers. It also has little impact on peer-to-peer performance, unless one of the peering devices are local.

8.2 VM migration

Since Device Lending decouples devices from their physical location, our KVM implementation makes it possible to shutdown, migrate and restart a VM on a different host in the cluster (cold migration). The guest will retain access to the same physical devices. We demonstrated this in VM evaluation (Sect. 7.3) and in our image classification workload (Sect. 7.5), where the OS image with all the installed software and device drivers resides on the same boot disk that is being used by the remote and local VM guests, the native remote host, and the native local host comparison.

With proper emulator support, it would also be possible to hot-add and hot-remove devices to a running VM instance. Using such hot-swap functionality, migrating a VM while it is running could be achieved by first removing all devices before migrating and then re-attaching them afterwards. However, this would temporarily disrupt their use and force guest drivers to reset all devices.

A strong candidate for future improvements is looking into real hot-migration techniques, remapping devices while they are in use and without (or with minimal) disruption. However, such a solution would be non-trivial. Not only would it require keeping memory consistent during the migration warm-up, but DMA TLPs could potentially be in-flight during the migration. A mechanism for rerouting TLPs without violating the strict ordering required by PCIe must be implemented, which most likely will require hardware-level support.

8.3 Security considerations

A VM may allocate several GB of memory, which may be scattered in physical memory. In order to conserve mapping resources, we use the IOMMU on the local system in order to provide linear continuous memory ranges that are trivially mapped over the NTB. However, pass-through uses the IOMMU in order to match I/O addresses with the guest physical memory layout. Furthermore, VFIO requires that passed-through devices are placed in an IOMMU domain per VM, in order to provide isolation. In our case, this is not possible since we already use the IOMMU, and the virtual device is in another domain.

However, we use the IOMMU on the *lender* instead to map I/O virtual addresses to guest physical memory layout and provide the necessary memory isolation. This guarantees that the device is only able to access the specific DMA windows to the VM it is assigned to, and the IOMMU on the borrower guarantees that the same windows can only be used to access the VM memory. Our solution therefore provides the same level of memory isolation as standard pass-through. It is also not possible for software running in the VM to access memory outside the device memory regions of assigned devices.

8.4 Interrupt forwarding

For VMs, we register an interrupt handler on the device-side lender and forward interrupts to the local borrowing system, as explained in Sect. 5. A benefit with this approach is that we are able to support all types of PCIe interrupts, legacy, MSI and MSI-X, while native Device Lending only supports MSI and MSI-X. However, this introduces additional latency and involves software handling on the lender.

An evaluation is needed to determine what impact increased latency for interrupts may have on the performance of device drivers. As this impact most likely is not negligible, a candidate for improvement is therefore to use the same approach as bare-metal Device Lending for MSI and MSI-X, and map these types of interrupts over the NTB. This would remove any special software handling other than on the borrowing system alone, where we still need to use the *eventfd* API in order to notify the VM.

9 Conclusion

In this paper, we presented how we have extended our Device Lending implementation with support for the KVM hypervisor, allowing pass-through of physically remote devices to local VM guests. By dynamically probing the available memory and fully supporting both MSI and MSI-X interrupts, we have greatly improved the usability of our previous Device Lending implementation [2]. With dynamic memory layout detection, it is possible to compose custom configurations of distributed I/O resources in a PCIe cluster, for both native and virtual machines. Our experimental evaluations prove that we are able to compose flexible configurations of remote devices and enable dynamic time-sharing of resources using Device Lending. Being able to scale by dynamically reassigning devices to machines that currently need them, makes it possible to support a flexible I/O infrastructure that meet processing requirements and at the same time makes it possible to optimize resource utilization.

We have also implemented support for borrowing multiple devices from different lenders and enabled peer-to-peer access between them, allowing remote I/O resources to be used as if they were attached to the same local fabric. This allows physically remote devices to be used by the local system, without requiring *any* modifications to either device drivers or applications and without adding *any* software overhead in the data path. As part of this evaluation, we have investigated the impact of I/O address virtualization on performance. Specifically, we have performed bandwidth and latency measurements for different data paths. By enabling peer-to-peer transfers and routing shortest path between devices, we demonstrate that native Device Lending does not add a performance overhead in the data path beyond what is expected for longer paths. However, our results indicate that a major performance bottleneck occurs when DMA read requests traverse the CPU root, as is the case when the IOMMU on the lender is enabled. The Intel Xeon CPUs used in our evaluation alters the requests in a way that leads to poor link utilization. This impacts our VM implementation, as it requires the use of device-side IOMMU in order to map the device to guest-physical address space. This warrants further evaluations of other CPU architectures.

We have also run a real-world medical imaging classification application with borrowed remote hardware resources. We compare a best-case bare-metal topology for local and remote devices, and show that we achieve close to local performance using Device Lending. We have also compared our newly implemented VM support to a local VM, and show that it is possible to run such a workload in a VM using remote physical devices. We argue that being able to run the exact same code using remote GPUs and hard disks as if they were locally installed, thus making use of a complex machine learning framework with one of the most complex GPU implementations on the market, demonstrate the strength of Device Lending.

Acknowledgements This work has been performed mainly in the context of the BIA project *PCIe* (#235530) funded by the Research Council of Norway, with contributions from the *LADIO* project (EU Horizon 2020 #731970). The authors would like to thank Stig Magnus Baugstø, Halvor Kielland-Gyrud, Roy Nordstrøm and Hugo Kohmann at Dolphin Interconnect Solutions. We also thank Kristoffer Robin Stokke for feedback on the manuscript.

References

1. Kristiansen, L.B., Markussen, J., Stensland, H.K., Riegler, M., Kohmann, H., Seifert, F., Nordstrøm, R., Griwodz, C., Halvorsen, P.: Device Lending in PCI Express Networks. In: Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV, pp. 10:1–10:6 (2016). https://doi.org/10.1145/2910642.2910650
2. Markussen, J., Kristiansen, L.B., Stensland, H.K., Seifert, F., Griwodz, C., Halvorsen, P.: Flexible device sharing in pcie clusters using device lending. In: Proceedings of the International Conference on Parallel Processing Companion, ICPP Companion, pp. 48:1–48:10 (2018). https://doi.org/10.1145/3229710.3229759
3. Fountain, T., McCarthy, A., Peng, F.: PCI express: an overview of PCI express, cabled PCI express and PXI express. In: Proceedings of International Conference on Accelerator & Large Experimental Physics Control Systems, ICALEPCS (2005)
4. Peripheral Component Interconnect Special Interest Group (PCI-SIG): PCI Express 3.1 Base Specification (2010). https://pcisig.com/specifications
5. Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., Weigert, J.: Intel virtualization technology for directed I/O. Intel Technol. J. **10**(03) (2006) https://doi.org/10.1535/itj.1003.02
6. Linux IOMMU Support. https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt
7. Nvidia Virtual GPU Technology (vGPU). http://www.nvidia.com/object/virtual-gpus.html
8. Peripheral Component Interconnect Special Interest Group (PCI-SIG): Single-root I/O Virtualization and Sharing Specification (2010). https://www.pcisig.com/specifications/iov/single-root/
9. Ravindran, M.: Extending Cabled PCI Express to Connect Devices with Independent PCI Domains. In: Proceedings of the IEEE Systems Conference, SysCon, pp. 1–7 (2008). https://doi.org/10.1109/SYSTEMS.2008.4519048
10. Regula, J.: Using Non-transparent Bridging in PCI Express Systems. PLX Technology Inc, Sunnyvale (2004)
11. Sullivan, M.J.: Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge. Specification, Intel Corporation (2010)
12. Saito, K., Anai, K., Igarashi, K., Nishikawa, T., Himeno, R., Yoguchi, K.: ATM bus system (1998)
13. Alnæs, K., Kristiansen, E.H., Gustavson, D.B., James, D.V.: Scalable coherent interface. In: Proceedings of International Conference on Computer Systems and Software Engineering, CompEuro, pp. 446–453 (1990). https://doi.org/10.1109/CMPEUR.1990.113656
14. The Case Against iWARP (2015). https://www.chelsio.com/wp-content/uploads/resources/iWARP-Myths.pdf
15. RoCE vs. iWARP Competitive Analysis (2017). http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf
16. Trivedi, A., Metzler, B., Stuedi, P.: A case for RDMA in clouds. In: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys, pp. 17:1–17:5 (2011). https://doi.org/10.1145/2103799.2103820
17. Huang, J., Ouyang, X., Jose, J., Wasi-Ur-Rahman, M., Wang, H., Luo, M., Subramoni, H., Murthy, C., Panda, D.K.: High-performance design of hbase with RDMA over InfiniBand. In: Proceedings of International Parallel and Distributed Processing Symposium, IPDPS, pp. 774–785 (2012). https://doi.org/10.1109/IPDPS.2012.74
18. Jiang, W., Liu, J., Jin, H.W., Panda, D.K., Gropp, W., Thakur, R.: High performance MPI-2 one-sided communication over InfiniBand. In: Proceedings of International Symposium on Cluster Computing and the Grid, CCGrid, pp. 531–538 (2004). https://doi.org/10.1109/CCGrid.2004.1336648
19. Duato, J., Pena, A., Silla, F., Mayo, R., Quintana-Ortí, E.: rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: Proceedings of International Conference on High Performance Computing and Simulation, HPCS pp. 224–231 (2010). https://doi.org/10.1109/HPCS.2010.5547126
20. Venkatesh, A., Subramoni, H., Hamidouche, K., Panda, D.K.: 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 (2014). https://doi.org/10.1109/HiPC.2014.7116875
21. Rosetti, D.: Benchmarking GPUDirect RDMA on Modern Server Platforms (2014). http://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/
22. Daglis, A., Novaković, S., Bugnion, E., Falsafi, B., Grot, B.: Manycore network interfaces for in-memory rack-scale computing. ACM SIGARCH Comput. Archit. News **43**(3), 567–579 (2015). https://doi.org/10.1145/2872887.2750415
23. Costa, P., Ballani, H., Razavi, K., Kash, I.: R2c2: a network stack for rack-scale computers. ACM SIGCOMM Comput. Commun. Rev. **45**(4), 551–564 (2015). https://doi.org/10.1145/2829988.2787492
24. Whitby-Strevens, C.: The transputer. ACM SIGARCH Comput. Archit. News **13**(3), 292–300 (1985). https://doi.org/10.1145/327070.327269
25. Hayes, J.P., Mudge, T., Stout, Q.F., Colley, S., Palmer, J.: A microprocessor-based hypercube supercomputer. IEEE Micro **6**(5), 6–17 (1986). https://doi.org/10.1109/MM.1986.304707
26. Peripheral Component Interconnect Special Interest Group (PCI-SIG): Multi-root I/O Virtualization and Sharing Specification (2008). https://www.pcisig.com/specifications/iov/multi-root/
27. Suzuki, J., Hidaka, Y., Higuchi, J., Baba, T., Kami, N., Yoshikawa, T.: Multi-root Share of Single-Root I/O Virtualization (SR-IOV) Compliant PCI Express Device. In: Proceedings of Symposium on High Performance Interconnects, HOTI, pp. 25–31 (2010). https://doi.org/10.1109/HOTI.2010.21
28. Tu, C.C., Lee, Ct, Chiueh, Tc: Secure I/O device sharing among virtual machines on multiple hosts. ACM SIGARCH Comput. Archit. News **41**(3), 108–119 (2013). https://doi.org/10.1145/2508148.2485932
29. Tu, C.C., Chiueh, T.c.: Seamless fail-over for PCIe switched networks. In: Proceedings of the International Systems and Storage Conference, SYSTOR, pp. 101–111 (2018). https://doi.org/10.1145/3211890.3211895
30. Dilk, P.: Microsemi Switchtec PAX: Advanced fabric gen3 pcie switch (2017). https://www.youtube.com/watch?v=OB7OuektR0E
31. Wong, H.: PCI express multi-root switch reconfiguration during system operation (2011)
32. VFIO—"Virtual Function I/O". https://www.kernel.org/doc/Documentation/vfio.txt
33. Jia, N., Wankhede, K.: VFIO Mediated Devices. https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt
34. Peripheral Component Interconnect Special Interest Group (PCI-SIG): Address Translation Services Revision 1.1 (2009). https://www.pcisig.com/specifications/iov/ats/
35. PXH830 Gen3 PCI Express NTB Host Adapter. http://www.dolphinics.no/products/PXH830.html
36. CUDA Toolkit Documentation v10.1.105 (2019). http://docs.nvidia.com/cuda/
37. Keras (2015). https://keras.io
38. TensorFlow: Large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/
39. Keras documentation: multi_gpu_model (2015). https://keras.io/utils/#multi_gpu_model
40. Borgli, R., Halvorsen, P., Riegler, M., Stensland, H.K.: Automatic hyperparameter optimization in keras for the mediaeval 2018 medico multimedia task. In: Working Notes Proceedings of the MediaEval 2018 Workshop (2018)
41. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR arXiv (2014). arXiv:abs/1409.1556
42. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: Proceedings of the Conference on Computer Vision and Pattern Recognition, CVPR (2009). https://doi.org/10.1109/CVPR.2009.5206848
43. Pogorelov, K., Randel, K.R., Griwodz, C., Eskeland, S.L., de Lange, T., Johansen, D., Spampinato, C., Dang-Nguyen, D.T., Lux, M., Schmidt, P.T., Riegler, M., Halvorsen, P.: KVASIR: A multi-class image dataset for computer aided gastrointestinal disease detection. In: Proceedings of the ACM Multimedia Systems Conference, MMSys, pp. 164–169 (2017). https://doi.org/10.1145/3083187.3083212
44. Hicks, S.A., Riegler, M., Pogorelov, K., Ånonsen, K.V., de Lange, T., Johansen, D., Jeppsson, M., Randel, K.R., Eskeland, S., Halvorsen, P.: Dissecting deep neural networks for better medical image classification and classification understanding. In: Proceedings of International Symposium on Computer-Based Medical Systems, CBMS (2018). https://doi.org/10.1109/CBMS.2018.00070
45. Pogorelov, K., Ostroukhova, O., Jeppsson, M., Espeland, H., Griwodz, C., de Lange, T., Johansen, D., Riegler, M., Halvorsen, P.: Deep learning and hand-crafted feature based approaches for polyp detection in medical videos. In: Proceedings of International Symposium on Computer-Based Medical Systems, CBMS, pp. 381–386 (2018). https://doi.org/10.1109/CBMS.2018.00073
46. Neugebauer, R., Antichi, G., Zazo, J.F., Audzevich, Y., López-Buedo, S., Moore, A.W.: Understanding PCIe performance for end host networking. In: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM, pp. 327–341 (2018). https://doi.org/10.1145/3230543.3230560

Jonas Markussen Jonas Markussen is a PhD student at Simula Research Laboratory, where his research is focused on new ways to use Non-Transparent Bridges in order to optimize data transfer paths and memory accessing by using their unique potential for mapping memory. Since 2018, Jonas has been working as a software architect for Dolphin Interconnect Solutions, continuing his work from his PhD. His research interests are distributed shared-memory applications, computer networks and cluster interconnects.

Lars Bjørlykke Kristiansen Lars Bjørlykke Kristiansen is a software architect at Dolphin Interconnect Solutions. He got his master's degree in Informatics at the University of Oslo, Norway in 2015 where his thesis laid the foundation for Device Lending. At Dolphin he continues his work on Device Lending, as well as exploring innovative new ways to exploit the unique shared memory capabilites of PCIe clusters and Non-Transparent Briding.

Rune Johan Borgli Rune Johan Borgli is a Ph.D. student at Simula Research Laboratory. He received his master's degree from the University of Oslo in 2018, where his master thesis topic was on hyperparameter optimization using Bayesian optimization on transfer learning for medical image classification. His research interests are machine learning workflows and pipelines, image processing, machine learning infrastructure optimization, and secure and privacy-oriented data handling. He is currently working on his Ph.D. thesis which will explore secure machine learning processing of privacy-sensitive data.

Håkon Kvale Stensland Håkon Kvale Stensland is a senior researcher at Simula Research Laboratory. He finished his master degree (MSc) in 2006 and received his doctoral degree (Ph.D.) in 2015 from the Department of Informatics, University of Oslo. At Simula, he is the deputy head of the Department of Advanced Computing and System Performance. From Simula, he is also leading the collaboration with Dolphin Interconnect Solutions, where we research sharing of GPUs and other IO devices between multiple machines connected in a PCI Express network. Håkon is also an adjunct associate professor at the University of Oslo, Department of Informatics, where he is involved in teachings and supervising Ph.D. and Master students.

Friedrich Seifert Friedrich Seifert obtained his master's degree in Computer Science (Dipl.-Inf.) from Chemnitz University of Technology, Germany, in 1999. He is working as Senior System and Software Architect for Dolphin Interconnect Solutions, where he focuses on developing innovative concepts for building compute and I/O clusters using Non-Transparent Bridging functionality found in state-of-the-art PCIe chipsets.

Michael Riegler Michael Riegler is a senior researcher at SimulaMet. He received his master's degree from Klagenfurt University with distinction and finished his PhD at the University of Oslo in two and a half years. His research interests are medical multimedia data analysis and understanding, image processing, image retrieval, parallel processing, crowdsourcing, social computing and user intent. He is involved in several initiatives like the MediaEval Benchmarking initiative for Multimedia Evaluation, which runs this year the Medico task (automatic analysis of colonoscopy videos). Furthermore he is part of an expert group for the Norwegian Council of Technology on Machine Learning for Healthcare reporting directly to the Norwegian Government.

Carsten Griwodz Carsten Griwodz is professor at the University of Oslo, Norway, and co-founder of ForzaSys AS, a social media startup for sports. He received his doctoral degree from Darmstadt University of Technology, Germany in 2000. His research interest is the performance of multimedia systems and his goal to understand how users can become sufficiently immersed in an experience depending on their goals and context. He explores research advances in fields ranging from operating system and networks to computer vision to understand and reach the point of sufficient immersion.

Pål Halvorsen Pål Halvorsen is a chief research scientist at SimulaMet, a professor at OsloMet University, a professor II at University of Oslo, Norway, and the CEO of ForzaSys AS. He received his doctoral degree (Dr.Scient.) in 2001. His research focuses mainly at distributed multimedia systems including operating systems, processing, storage and retrieval, communication and distribution from a performance and efficiency point of view. He also is a member of the IEEE and ACM.

Paper V

SmartIO: Zero-overhead Device Sharing through PCIe Networking

Authors: **Jonas Markussen**, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, Carsten Griwodz.

Abstract: The large variety of compute-heavy and data-driven applications accelerate the need for a distributed I/O solution that enables cost-effective scaling of resources between networked hosts. For example, in a cluster system, different machines may have various devices available at different times, but moving workloads to remote units over the network is often costly and introduce large overheads compared to accessing local resources. To facilitate I/O disaggregation and device sharing among hosts connected using Peripheral Component Interconnect Express (PCIe) non-transparent bridges, we present SmartIO. NVMes, GPUs, network adapters, or any other standard PCIe device may be borrowed and accessed directly, as if they were local to the remote machines. We provide capabilities beyond existing disaggregation solutions by combining traditional I/O with distributed shared-memory functionality, allowing devices to become part of the same global address space as cluster applications. Software is entirely removed from the data path, and simultaneous sharing of a device among application processes running on remote hosts is enabled. Our experimental results show that I/O devices can be shared with remote hosts, achieving native PCIe performance. Thus, compared to existing device distribution mechanisms, SmartIO provides more efficient, low-cost resource sharing, increasing the overall system performance.

Candidate's contributions: The ideas for the device-oriented extension to the SISCI API grew out from Markussen's experiences with implementing MDEV for Paper III and Paper IV. Markussen contributed with several new ideas for the design of this API, and implemented these. He collaborated on the effort of combining these ideas with previous work into the complete SmartIO system. Furthermore, Markussen came up with the idea for, designed, and implemented the prototype distributed NVMe driver using this API extension, including the queue offloading idea and support for running the driver on GPUs. Limitations in the initial API design were uncovered during this process, and Markussen made subsequent improvements to both design and implementation of the API throughout the development of the driver. Additionally, he designed and implemented several workloads for this NVMe driver using GPUs and Device Lending in order to demonstrate the novelty and completeness of the SmartIO solution, as well as the performance benefits. He conducted a thorough and exhaustive performance analysis of all components of the SmartIO system, as well as evaluating the entire system, and investigated and implemented solutions for eliminating performance overheads in the system. Finally, Markussen wrote most of the text for the paper, and also wrote the necessary tools and benchmarking programs for the evaluation, including the FIO integration for the NVMe driver and implementing GPU and NVMe test programs.

Published in: *Transactions on Computer Systems.* ACM. Published online June 2021, issue date July 2021, volume 38, issue 1-2, article 2, pp. 2:1–2:78.

DOI: 10.1145/3462545

Contributed to: All objectives (Objectives 1–6).

SmartIO: Zero-overhead Device Sharing through PCIe Networking

JONAS MARKUSSEN and LARS BJØRLYKKE KRISTIANSEN, Dolphin Interconnect Solutions, Norway
PÅL HALVORSEN, SimulaMet, Norway
HALVOR KIELLAND-GYRUD, Dolphin Interconnect Solutions, Norway
HÅKON KVALE STENSLAND, Simula Research Laboratory, Norway
CARSTEN GRIWODZ, University of Oslo, Norway

The large variety of compute-heavy and data-driven applications accelerate the need for a distributed I/O solution that enables cost-effective scaling of resources between networked hosts. For example, in a cluster system, different machines may have various devices available at different times, but moving workloads to remote units over the network is often costly and introduces large overheads compared to accessing local resources. To facilitate I/O disaggregation and device sharing among hosts connected using Peripheral Component Interconnect Express (PCIe) non-transparent bridges, we present SmartIO. NVMes, GPUs, network adapters, or any other standard PCIe device may be borrowed and accessed directly, as if they were local to the remote machines. We provide capabilities beyond existing disaggregation solutions by combining traditional I/O with distributed shared-memory functionality, allowing devices to become part of the same global address space as cluster applications. Software is entirely removed from the data path, and simultaneous sharing of a device among application processes running on remote hosts is enabled. Our experimental results show that I/O devices can be shared with remote hosts, achieving native PCIe performance. Thus, compared to existing device distribution mechanisms, SmartIO provides more efficient, low-cost resource sharing, increasing the overall system performance.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; *Cloud computing*; • **Hardware** → *Buses and high-speed links*; • **Software and its engineering** → *Distributed memory*; *Distributed systems organizing principles*; • **Information systems** → Distributed storage;

Additional Key Words and Phrases: Resource sharing, composable infrastructure, I/O disaggregation, PCIe, cluster architecture, Device Lending, NVMe, GPU, NTB, distributed I/O

J. Markussen is also with Simula Research Laboratory, Norway.
P. Halvorsen also with Oslo Metropolitan University, Norway.
H. K. Stensland is also with University of Oslo, Norway.
C. Griwodz is also with SimulaMet, Norway.
Authors' addresses: J. Markussen, L. B. Kristiansen, and H. Kielland-Gyrud, Dolphin Interconnect Solutions AS, Nils Hansens vei 13, 0667 Oslo, Norway; emails: {jonas, larsk, halvor}@dolphinics.com; P. Halvorsen, Simula Metropolitan, Pilestredet 52, 0167 Oslo, Norway; email: paalh@simula.no; H. K. Stensland, Simula Research Laboratory, PO Box 134, 1325 Lysaker, Norway; email: haakonks@simula.no; C. Griwodz, Department of Informatics, University of Oslo, PO Box 1080, Blindern, 0316 Oslo, Norway; email: griff@ifi.uio.no.

0734-2071/2021/06-ART2
https://doi.org/10.1145/3462545

ACM Reference format:
Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. 2021. SmartIO: Zero-overhead Device Sharing through PCIe Networking. *ACM Trans. Comput. Syst.* 38, 1-2, Article 2 (June 2021), 78 pages.
https://doi.org/10.1145/3462545

1 INTRODUCTION

High-performance computing workloads often have high requirements for I/O resources. For example, many computing clusters rely on compute accelerators, such as **graphics processing units (GPUs)** and **field-programmable gate arrays (FPGAs)**, to increase the processing speed. Moving data efficiently between networked nodes and onto such compute accelerators has been a research challenge for decades. In recent years, we have also seen a convergence of high-performance computing, big data, and machine learning research fields. This has led to new demands to I/O performance where distributed, high-volume storage is becoming a requirement for high-performance computing, while low latency networking and facilitating access to compute accelerators have become cloud computing issues [16, 80, 84]. If I/O resources (devices) are distributed scarcely among hosts, then cluster nodes with I/O resources may become bottlenecks when a workload requires heavy computation on GPUs or fast access to storage. Contrarily, over-provisioning nodes with resources may lead to devices becoming underutilized if the workload's demands are more sporadic. Heterogeneous workloads may even require widely different compositions of devices for individual nodes. Being able to share and dynamically partition devices between nodes in a cluster leads to more efficient utilization, as I/O resources can be scaled up or down based on current workload requirements.

In cloud computing environments, such dynamic scaling and resource partitioning is often handled through virtualization. **Virtual machine (VM)** hypervisors may dynamically add virtual I/O devices to VM instances on demand. It is even possible to temporarily suspend computation to migrate VMs to hosts with more hardware resources, should the VM's requirements exceed the available local resources. However, resource virtualization may not be viable when the raw, bare-metal I/O performance is required, for example in the case of GPU-intensive machine learning workloads. In this regard, it is possible to "pass through" physical I/O devices to a VM guest using an **I/O Memory Management Unit (IOMMU)**. The IOMMU facilitates direct access to hardware from the guest without compromising the virtualized environment. Although pass-through allows physical hardware to be used with minimal software overhead, this technique suffers from a lack of flexibility as the physical devices are tightly coupled with the hosts they are installed in. Distributing VMs across hosts in the network in a way that maximizes resource utilization and adapts dynamically to varying I/O requirements, without sacrificing the bare-metal performance that pass-through provides, remains a challenge.

Another challenge is the networking technology itself. Many network adapters support zero-copy of application memory from one system to another through **remote direct memory access (RDMA)** [32]. RDMA is not only used in many distributed shared-memory cluster applications, but is also frequently used for implementing resource disaggregation. Low-latency storage devices, such as **non-volatile memory express devices (NVMes)**, can be shared at the block-level in the cluster. This is the case for **NVMe over Fabrics (NVMe-oF)** [29], where RDMA is used to provide direct access and avoid going through the block-layer on the **operating system (OS)** on the server. Similarly, the result of a GPU computation may be copied out of GPU memory and onto the network directly using RDMA, without being copied to system memory first and going through the network stack [91]. RDMA disaggregation is usually implemented as application-specific middleware. Although this often requires application software to use specific programming models and semantics, such as message-passing, the benefit is that resources may be shared by several hosts in the network. However, while RDMA allows data to be transferred efficiently over the network, translation between the network protocol and the local I/O bus is unavoidable. Compared to accessing a local device, this protocol translation incurs latency overheads that are not insignificant.

Peripheral Component Interconnect Express (PCIe) is the most widely used standard for connecting devices to a computer system. Although it was originally designed as a local I/O bus connecting devices to the **central processing unit (CPU)** on a motherboard, extending the PCIe bus out of a single computer and connecting several systems is made possible by using a special type of device called **non-transparent bridge (NTB)**. NTBs can be embedded as a CPU feature [77, 95], but are more commonly implemented in PCIe switch chips [13, 82], allowing independent computer systems to interconnect with plug-in host adapter cards and external cables [44, 50, 67, 69]. Unlike other interconnection technologies, solutions built with PCIe networking allow resources to be accessed with very little performance overhead as no protocol translation is required. However, while some disaggregation approaches using NTBs have been proposed in the past [31, 89], these implementations present solutions where devices are owned by a dedicated server. As distributing resources is generally only possible to hosts that are directly connected to the same switch as this server, these approaches forgo the flexibility of fully distributed cluster computing systems. Alternative PCIe-based solutions rely on additional virtualization functionality in the PCIe switch chip hardware to partition the PCIe fabric and create virtual device trees for each individual host [15, 51]. These solutions allow devices to be directly attached a switch rather than a server. However, these solutions are only able to disaggregate resources at the device level. Sharing the same device with multiple hosts either requires virtualization support in the device itself, i.e., **Single-Root I/O Virtualization (SR-IOV)**, or additional distribution methods, such as RDMA.

To address these challenges, we present our *SmartIO* system for sharing resources and distributing devices in a heterogeneous, PCIe-interconnected cluster. Unlike existing solutions, our system is able to provide sharing and disaggregation capabilities at multiple abstraction levels: distributing devices to physical hosts, distributing devices to VMs, and enabling disaggregation of devices and memory in software. In addition, our SmartIO system is fully distributed. We avoid relying on dedicated servers and instead allow all hosts to contribute their own local resources and access remote resources, even at the same time. This blurs the distinction between remote and local resources, and scaling out and increasing the overall I/O resource utilization in the system becomes easier.

SmartIO is implemented on top of the inherent memory mapping capabilities of NTBs, allowing cluster nodes to map parts of the address space in remote hosts. Our system effectively makes all hosts, including their internal resources (both devices and memory), part of a common PCIe domain. Remote resources can be accessed directly over native PCIe, without requiring any software in the data path or network protocol translation. Furthermore, by relying on PCIe shared-memory techniques, SmartIO is able to abstract away the physical location of devices and memory resources. Our implementation translates memory addresses between different address domains and resolves paths through the PCIe network in a manner that is transparent to both application software and device drivers. As all nodes may contribute their resources, and not only dedicated servers, our SmartIO is able to provide optimizations based on resource locality and minimizing data movement, without requiring the user to be aware of the underlying PCIe topology. This unlocks a new potential in PCIe-connected cluster systems, as application software no longer needs to be written with accessing remote resources in mind, but can be implemented as if resources are local.

We have previously demonstrated how Device Lending allows devices to be dynamically assigned to different machines, making it possible for a system to access remote PCIe devices as if they were locally installed [41]. We have also shown how our Device Lending method extends to VMs by implementing a **mediated device interface (MDEV)**, which facilitates pass-through of remote PCIe devices to VMs running on any host in the cluster [48, 49]. Our new complete SmartIO sharing solution does not only incorporate this earlier work, but greatly extends and supersedes it. We have generalized the core components of our original Device Lending implementation, i.e., the mechanism that enables direct access over PCIe in a manner that is transparent to both device and device driver, and have developed an entirely new **application programming interface (API)**. This new API provides device driver functionality to shared-memory cluster applications, such as mapping shared memory regions for **direct memory access (DMA)** from the device and memory-mapping device registers into application address space. By making device operation part of distributed cluster applications and allowing devices to access shared memory regions using native DMA, it becomes possible to disaggregate devices in software. As such, our new API enables *simultaneous sharing* of devices between software processes running on different hosts in the cluster, in addition to device-level distribution capabilities provided by Device Lending and MDEV.

In short, SmartIO is a flexible framework for device distribution and resource sharing that enables cost-effective scaling of resources between PCIe-networked hosts. The main contributions of our work are listed as follows:

- We have incorporated our previous Device Lending method into our complete SmartIO solution. NVMes, GPUs, network adapters, and any standard PCIe device can be distributed to remote systems and used without any performance difference compared to local access. Devices appear as if they are dynamically hot-added to the system, and can be used by existing application software and device drivers without requiring any modifications.
- SmartIO also includes our MDEV extension to Device Lending. This interface extends the Linux **Kernel-based Virtual Machine hypervisor (KVM)**. Our extension facilitates direct access to remote physical devices for VM guests, allowing VMs to run on any host in the network and use (remote) devices with bare-metal performance.
- We have created a new device-oriented API for writing device drivers as shared-memory applications. This makes it possible to disaggregate devices in software, similarly to RDMA disaggregation solutions. Unlike RDMA, however, resources are accessed over native PCIe, which allows resources to be shared without introducing a performance penalty. Through our API, device driver implementations may take full advantage of PCIe shared memory capabilities, such as remote memory access and multicasting, without requiring awareness of the PCIe topology and the different address domains of remote systems. This makes it easier for application software to optimize data flow through the PCIe network.
- We have developed a prototype NVMe device driver using our new device-oriented API. Although the Device Lending component of SmartIO makes it possible to use existing device drivers, most device drivers are written in a way that assumes exclusive control over the device. Using Device Lending alone, a device may only be used by a single host at the time. To demonstrate software-enabled disaggregation, we have implemented a *distributed* NVMe driver. As a proof of concept, we show a single NVMe device can be shared and operated by 30 cluster nodes simultaneously, without requiring SR-IOV. This driver also demonstrates how multiple sharing aspects of our system may be combined, by disaggregating (remote) GPU memory and enabling memory access optimizations.
- To prove that our solution enables zero-overhead sharing, we provide a comprehensive performance evaluation covering all components of our SmartIO solution, including our earlier

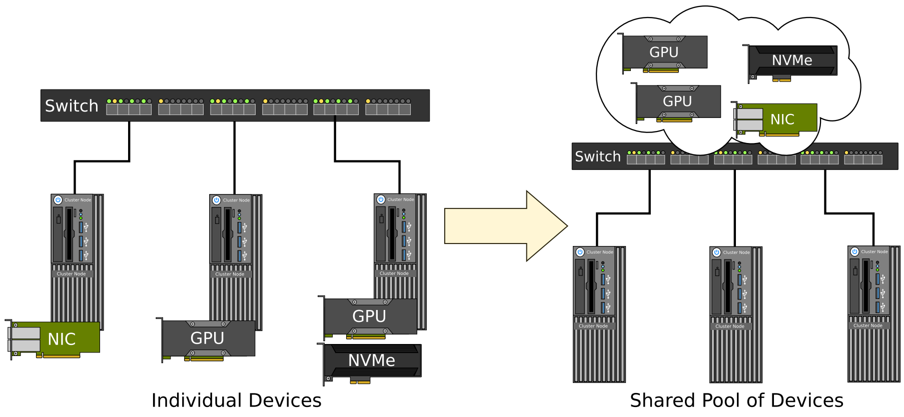


Fig. 1. SmartIO allows the internal devices of hosts in the network to be shared with other hosts connected to the same fabric. Nodes in a PCIe-networked cluster can contribute their internal devices to a shared device pool, and borrow resources from that pool when needed.

> Device Lending and MDEV work. We have performed entirely new experiments, using both synthetic microbenchmarks and realistic large-scale workloads. Our experimental results confirm that I/O devices can be distributed to, and shared with, remote hosts, without any performance penalty beyond what is expected for longer PCIe paths. In fact, all our experiments prove that remote devices can be used *without any performance overhead* compared to local access in terms of latency and throughput.

The rest of this article is structured as follows: Section 2 gives a high-level overview of our SmartIO system. Section 3 explains the basic building blocks of shared-memory networking with PCIe. In Section 4, we detail our Device Lending method, and in Section 5, we explain how the original Device Lending was enhanced with hypervisor support (MDEV). In Section 6, we describe our new software API and use a distributed NVMe driver implementation as an example implementation. We present our experimental results and extensive evaluation in Section 7, before we provide a discussion of other aspects and considerations of our SmartIO solution in Section 8. Finally, we put the work in the context of state of the art in Section 9, and conclude the article in Section 10.

2 SYSTEM OVERVIEW

Our SmartIO solution allows the local resources of a host, i.e., memory and devices, to be accessed directly by remote hosts, over standard PCIe. SmartIO works for *all* standard PCIe devices. Individual device functions of multi-function devices may be distributed to different hosts in the network, or to the same host should it require multiple resources. It is even possible to disaggregate a single device (function) in software, and distribute it to multiple hosts.

As depicted in Figure 1, we can imagine this as hosts contributing their internal resources to a pool of shared resources. Through a process of borrowing devices and releasing them when they are no longer needed, it is possible to support a dynamic and composable I/O infrastructure consisting of a combination of local and remote resources. Whether devices are actually local or remote becomes irrelevant to the user, as SmartIO eliminates this distinction, both function and performance wise. In other words, SmartIO is a solution for scaling out and using more hardware resources than there are available in a single host.

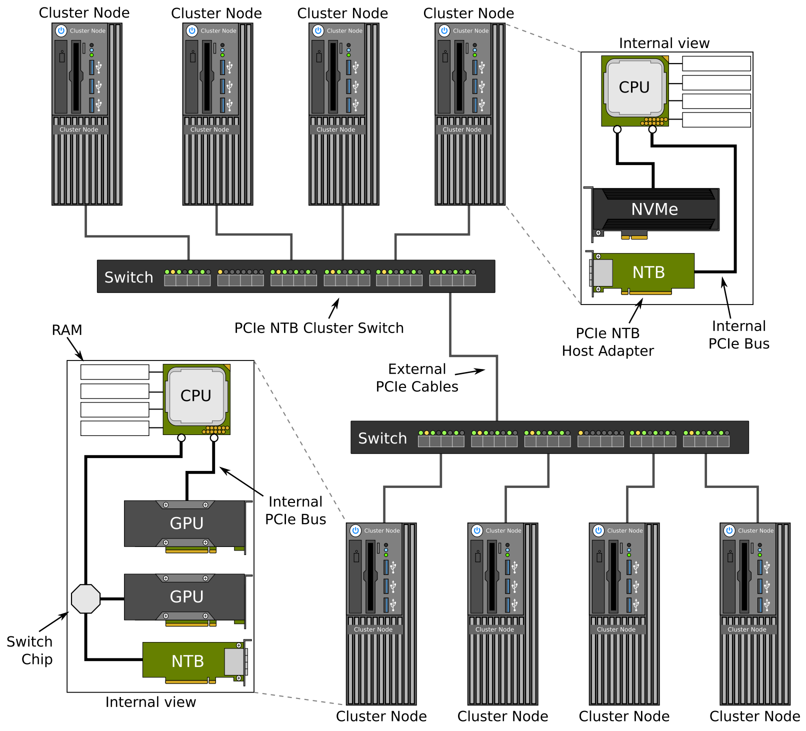


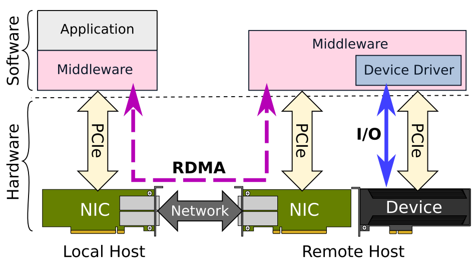
Fig. 2. We can create a heterogeneous PCIe cluster by interconnecting nodes (hosts) with external PCIe links using adapter cards capable of non-transparent bridging (NTB). In such clusters, the CPUs as well as the internal devices of each node are all attached to the same PCIe network fabric.

2.1 Motivation and Challenges

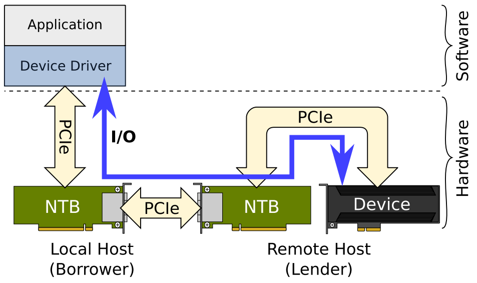
Due to its very low latency overhead and memory addressing properties, using PCIe as a high-speed interconnection technology is a compelling alternative to traditional networking technologies [44, 50, 67]. However, because PCIe was originally designed as a local I/O bus, connecting devices to the CPU on a motherboard, individual computer systems operate with different PCIe address domains. Interconnecting systems using PCIe require translating memory transactions from one address domain to another. The most common method of translating addresses is to use NTBs [69, 82, 87]. Figure 2 illustrates how several computer systems may be interconnected in a cluster, by implementing adapter cards and cluster switches with NTBs. The inherent memory address translation capabilities of NTBs make it possible to map (parts of) the address space of remote systems. More interesting, however, is the fact that in such PCIe networks, both CPUs and internal PCIe devices are attached to the same, shared PCIe fabric.

Remote resources, such as memory and I/O devices, can be mapped into a local system and accessed through the NTB. Similarly, a remote device capable of DMA may also use the NTB to access local resources. This eliminates the need to use memory on the remote node as an intermediate step when transferring data. As illustrated in Figure 3, software overhead can be avoided, since all memory address translations can be done in NTB hardware.

However, setting up such NTB mappings requires awareness of the address space on the remote system. When initiating DMA transfers, a device driver must use addresses that corresponds with



(a) Accessing remote resources using RDMA.



(b) Accessing remote resources using SmartIO.

Fig. 3. Many disaggregation solutions have performance overheads, because they rely on middleware or other forms of software facilitation on the remote system. Using SmartIO, remote hardware can be accessed directly without any software in the critical path by setting up memory mappings over the NTB.

the remote device's address space to enable a DMA-capable device to read or write across the NTB. This greatly increases the programming complexity of device drivers. Therefore, our SmartIO system provides a mechanism for using NTBs while remaining agnostic about the address space in remote systems. The physical location of a resource, as well as the address space layout in the host it is installed in, is entirely abstracted away.

Nevertheless, this abstraction gives rise to another challenge; a device driver that is unaware that a device is remote may assume that the entire local address space can be reached by the device. It is generally not possible to predict in advance which memory addresses a device driver may use, yet NTB mappings must be in place before the device driver initiates DMA transfers. Deferring mappings until the device driver initiates DMA would require synchronizing with the remote system in the critical path, thus increasing the overall latency. A naive workaround is mapping the entire memory for the device, but this solution does not scale for multiple hosts. SmartIO solves this, and is able to prepare necessary memory-mappings in advance, without introducing any communication overhead in the critical path.

2.2 Overall Design

Our system is composed of *"borrowers"* and *"lenders."* A lender is a computer system that registers one or more of its internal PCIe devices with SmartIO, allowing the devices to be distributed to and used by remote hosts. A borrower is a system that is currently using such a device. While a device only has one lender, namely, the computer system where it is physically installed, there can be several borrowers using it simultaneously.[1] SmartIO also makes it possible for a system to act as both lender and borrower at the same time, lending out its own local devices and simultaneously borrowing remote devices from other hosts.

Building PCIe networking into our system is a crucial part of our design, as it enables access to remote resources with very low latency and extremely low computing overheads. The hard separation between local and remote is blurred, with regard to both functionality and performance. Furthermore, this design means that the implementation complexity of SmartIO lies in software. SmartIO can be implemented for existing computer systems that are connected with NTBs, using either on-board PCIe switch chips or plug-in adapter cards, in any network topology.

[1]Note that the term "borrower" is not always synonymous with the physical host using the device in every context, but may refer to an individual software process or a VM.

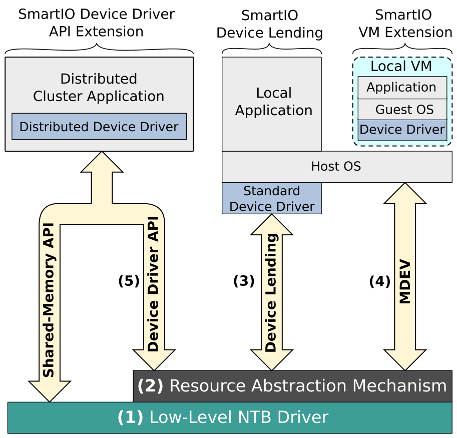


Fig. 4. SmartIO provides different interfaces that facilitate access to a remote resource. These interfaces present an abstraction layer to application software and device drivers, providing a logical decoupling of devices and which physical hosts they are installed in.

Figure 4 illustrates the different components of our system and how they fit together:

(1) **Low-level NTB driver:** Our SmartIO solution is built on top of NTB interconnection technology. The low-level NTB driver makes it possible to connect hosts over a PCIe network fabric and set up memory-mappings on demand. Moreover, the NTB driver also enables individual systems to contribute parts (or "segments") of their local memory to a cluster-wide, distributed shared-memory space. Cluster applications may use the **Software Infrastructure Shared-Memory Cluster Interconnect API (SISCI)** [22] to manage local and remote segments of memory and map them into the application's local address space.

(2) **Resource abstraction mechanism:** SmartIO provides functionality for transparently translating I/O addresses between different address domains, resolving paths in the cluster, and dynamically setting up necessary NTB mappings for the borrowing system and the device. This makes it possible to abstract away the location of the device, i.e., which host machine it is installed in, in a manner that is transparent to both the device and the software process using the device. With this abstraction, SmartIO can facilitate the use of remote resources (both memory and devices) without requiring software to be aware of the underlying, physical PCIe topology or the internal I/O address space layout of remote hosts. SmartIO also supports setting up mappings between multiple devices, even when they reside in different lenders, allowing PCIe transactions between them to be routed along the shortest path in the PCIe network (peer-to-peer).

(3) **Device Lending:** SmartIO incorporates our Device Lending method [41], which allows devices to be time-shared among hosts in the PCIe network. By borrowing a device and inserting it into the local device tree, the remote device appears to be hot-added to a local system. Devices can, therefore, be dynamically added to the system, without requiring the borrowing host to reboot. When the host performs configuration cycles and sets up memory mappings, SmartIO is able to intercept this and inject resolved remote addresses. This allows existing software to use our system without requiring any modifications or special adaptions; device drivers, application software and even the OS can use the device as if it was locally installed. While Device Lending only allows devices to be distributed to a single host at the time, it is nevertheless highly suitable in the case where a device has a complex or proprietary device driver, and using existing drivers is the only viable option for operating the device.

(4) **MDEV:** Our MDEV extension to the KVM hypervisor [48, 49] facilitates pass-through of borrowed devices to VMs running on the host. VM guests can access these devices directly without breaking out of the memory isolation provided by the virtualization, even when the devices are remote. This allows VMs to be distributed on different hosts in the cluster while benefiting from the bare-metal performance of direct access to physical hardware.

(5) **Device driver API:** As an alternative to Device Lending and MDEV, our SmartIO solution also provides a new device driver API extension for managing devices and developing distributed device drivers using cluster functionality. This new contribution extends the existing SISCI API with programming semantics for memory-mapping device registers and making shared memory segments available for a DMA-capable device. Device operation becomes part of the cluster application itself, allowing devices to access shared memory segments using native DMA. Furthermore, by relying on our SmartIO system to resolve memory addresses between the individual address domains, a driver implementation does not need to consider the system-local address space of the cluster node where the device is installed. This greatly reduces the complexity of implementing distributed applications, as it becomes possible for software to assume that resources are local, while taking full advantage of PCIe-based shared memory capabilities. Using this API extension, devices may be disaggregated at the software level and shared simultaneously between application processes running on different remote hosts.

Finally, it should be noted that the design of our system enables sharing at multiple abstraction levels. It is possible to combine the different interfaces of SmartIO. For example, using our API extension, we can disaggregate the device memory of a remote GPU being borrowed with Device Lending, even if it is managed by a proprietary device driver that is unaware that the device is remote.

3 PCIE-INTERCONNECTED CLUSTERS

While there are several networking technologies that make it possible to build clusters of networked computers, such as Infiniband, 100/200 Gigabit Ethernet, and Fibre Channel, PCIe is interesting in that connecting multiple systems with PCIe will also connect their internal devices to the same interconnection fabric. The idea of a unified bus for the inner components of a computer to connect the devices with the other cluster machines, however, is not new. It was already imagined for both ATM [72] and SCI [6]. Nevertheless, these ideas never got implemented, because neither technology were picked up as an internal interconnection network for computers. In contrast, PCIe is today the most widely adopted standard for connecting devices in a system [25].

The most common way of extending the PCIe bus out of a single system to connect several systems to the same PCIe fabric, is by using special devices called NTBs [50, 67, 69, 87, 89]. By implementing NTBs as a peripheral device, independent computer systems can interconnect with plug-in adapter cards and external cables. Using such adapter cards and cluster switches with NTB-capable ports, we have created a heterogeneous PCIe cluster, supporting up to 60 PCIe-networked nodes.

3.1 PCIe Endpoints

PCIe is a high-speed serial computer expansion bus standard and uses point-to-point links, where a link consists of 1 to 16 lanes. Each lane is a full-duplex serial connection. Data is striped across multiple lanes, so broader links yield higher bandwidth. PCIe revision 3.1 (Gen3) [61] allows a theoretical maximum bandwidth of 15.75 GB/s for an x16 link — approximately 13.8 GB/s of usable throughput.

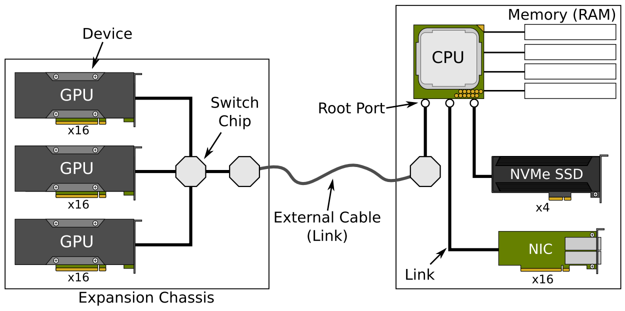


Fig. 5. Example of a PCIe topology using an external link to connect an expansion chassis to a computer system. The devices in the expansion chassis are part of the same PCIe tree as the internal devices, because all downstream links (including the external cable) are *transparent*.

As illustrated in Figure 5, a PCIe domain is structured as a tree. At the top of the tree, we have the "root ports," acting as the connection between the PCIe fabric and the CPU. This forms what is known as a "root complex." Devices are the leaf nodes in the PCIe domain, and are known as "endpoints" in PCIe terminology.

Some PCIe devices may support multiple functions, which appear to the system as a group of distinct devices, each with a separate set of resources and device memory regions. The term "device" actually refers to an individual function. An example of a multi-function device is a multi-port Ethernet adapter, where individual ports can be implemented as separate functions, or a GPU with a sound device, where the video controller appears as one device and the sound card as another. It is also possible for a device to implement SR-IOV [62]. SR-IOV-capable devices appear to the system to have multiple (virtual) functions. Note that our SmartIO system makes no distinction between physical and virtual functions.

3.2 Address-based Routing

The defining feature of PCIe is that devices are mapped into the same address space as the CPU and system memory, as depicted in Figure 6. Because this mapping exists, a CPU can read and write to device memory the same way it would access system memory.[2] Likewise, if a device is capable of **direct memory access (DMA)**, then it can read from and write to system memory. A device may even access other devices on the fabric, as they too are mapped into the same address space.

This mapping occurs when a system enumerates the PCIe tree and accesses the configuration space of each device attached to the fabric. The configuration space contains a description of the capabilities of the device, such as the device's memory regions. The system will reserve a memory address range for each of the device's memory regions. The start addresses are then written to the device's **Base Address Registers (BARs)** in its configuration space, making the device aware of the address space mapping. Therefore, the term "BAR" is synonymously used for device memory regions, and a device may have up to six BARs.

Like other networking technologies, PCIe also uses a layered protocol. The physical layer and data link layer are responsible for flow control, error correction and signal encoding. The uppermost layer is called the transaction layer, and its responsibility includes forwarding memory reads and writes as "transactions." Such transactions are routed in the PCIe fabric based on their

[2]This is often referred to as memory-mapped I/O (MMIO).

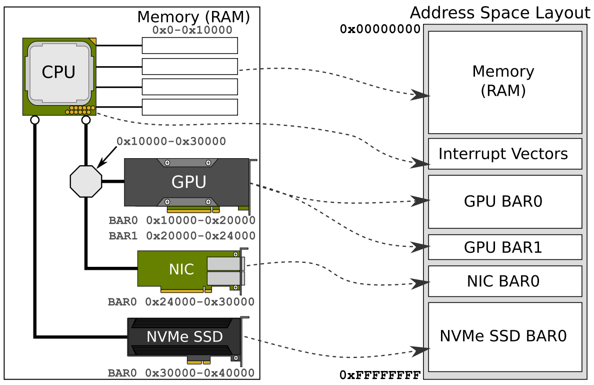


Fig. 6. Device memory regions (BARs) are mapped into the same address space as system memory.

addresses. The transaction layer is also responsible for packet ordering, ensuring that memory operations in PCIe are strictly ordered.[3]

In Figure 5, we also illustrate how the PCIe tree may be extended through the use of an expansion chassis. Devices in an expansion chassis are connected to the same root complex (CPU) through a series of transparent switches. These switches form subtrees in the network. During the enumeration, switch ports are assigned the combined address range of their downstream devices (Figure 6). This allows memory transactions to be routed hierarchically in the PCIe tree where memory transactions are forwarded either upstream or downstream based on the address. An invariant of this hierarchical routing is that memory accesses do not need to pass through the root, but can be routed using the shortest path. This is referred to as "peer-to-peer" in PCIe terminology. In Figure 5, the internal switch in the expansion chassis will have the combined downstream address range of all three GPUs, allowing memory accesses to be routed directly between them. Some PCIe switch chips also support multicasting, allowing memory writes to be replicated to multiple selected ports in a single operation [61].

PCIe also uses **message-signaled interrupts (MSI)** instead of physical interrupt lines. MSI-capable devices post a memory write to the CPU, using a specific address and payload assigned by the system. The memory write is then interpreted by the CPU, which uses the payload and address to raise an interrupt. MSI-X is an extension to MSI, allowing up to 2048 different interrupt vectors. A benefit of this is that an MSI-X interrupt can target a specific CPU core on multi-core systems. Additionally, separate MSI-X vectors can be used to indicate different types of events.

3.3 Non-transparent Bridging

As PCIe tree enumeration and address reservation is typically done during system start up, the address space layout will vary from system to system. Different systems, or different root complexes, will have independent address space layouts. Because of this, a PCIe domain has exactly *one* active root complex at any point in time. Two independent CPUs are not allowed to coexist in the same domain. However, by using an NTB implementation [44, 69, 82], two root complexes, meaning independent hosts, can be connected together over PCIe. Although not formally standardized, NTBs are a widely adopted solution, and all NTB implementations have similar capabilities [87]. NTBs

[3]The PCIe standard also specifies optional support for relaxed ordering, but strict ordering is mandatory and used by default.

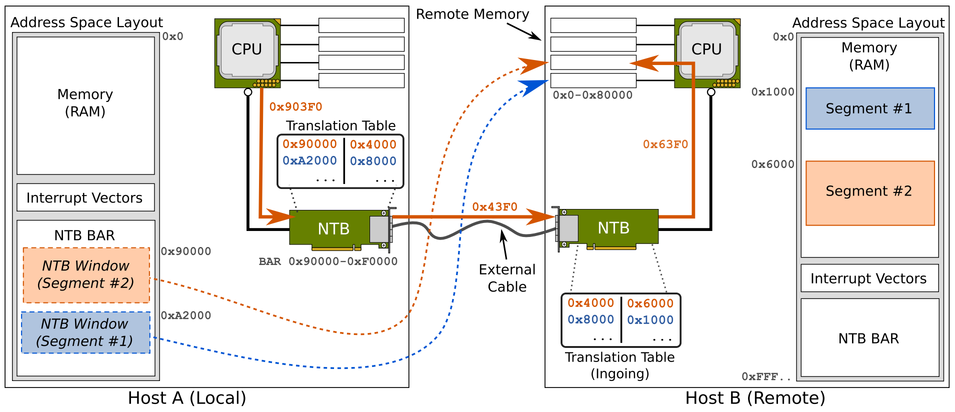


Fig. 7. Example of two independent PCIe root complexes connected together using an NTB. The link between the two hosts is *non-transparent*, and the NTB translates addresses between the two domains. Host A has mapped parts, or segments, of Host B's memory through its local NTB, providing Host A with "windows" into the remote system's address space.

can be embedded as a CPU feature, such as Intel Xeon [77] and AMD Zeppelin [95], but are more commonly implemented in PCIe switch chips [13, 82].

Figure 7 depicts two independent root complexes connected using NTB adapter cards with an external PCIe cable. Despite the name, an NTB actually appears as a PCIe *endpoint*. Just like regular endpoints, NTBs appear to have one or more memory regions, or BARs, that are reserved and mapped by the system during the enumeration. However, instead of being backed by memory or device registers, reads and writes to these memory regions will be forwarded from one side of the NTB to the other, translating the memory addresses in the process. As these memory regions appear to the system as any other memory-mapped device memory region, a local CPU can read from or write to them as if it was local device memory.

Note that the address space associated with the NTB BAR may be too small to cover all system memory of the remote system. While it is possible to adjust the BAR sizes and provide larger ranges, many systems do not support support large device memory regions. However, NTB implementations also support dividing their range into "windows." By using a different base offset per NTB window, it is possible to map arbitrary ranges of the remote system's address space. Such offset mappings makes it is possible to map different parts of a remote system's address space into local address space. The far-side address of a mapping is stored in a look-up table, making the address translation between the two domains very fast. However, the number of NTB windows is limited by the number of entries in the look-up table.

The SISCI shared memory API [22] provides functionality for allocating linear "segments" from a pool of contiguous memory pages that is reserved by the low-level NTB driver in advance. These linear segments can be "exported," allowing remote hosts to map them through their NTBs and access it as if it was local device memory. By allowing segments of their own local memory to be mapped by remote hosts, individual nodes effectively contribute to a distributed shared-memory architecture comprised of such memory segments. Multiple nodes may even map the same memory segment. By using the SISCI API, these memory segments can be mapped into the virtual address space used by application processes running on different nodes. This allows distributed applications to read and write to shared memory segments as is if it was local memory.

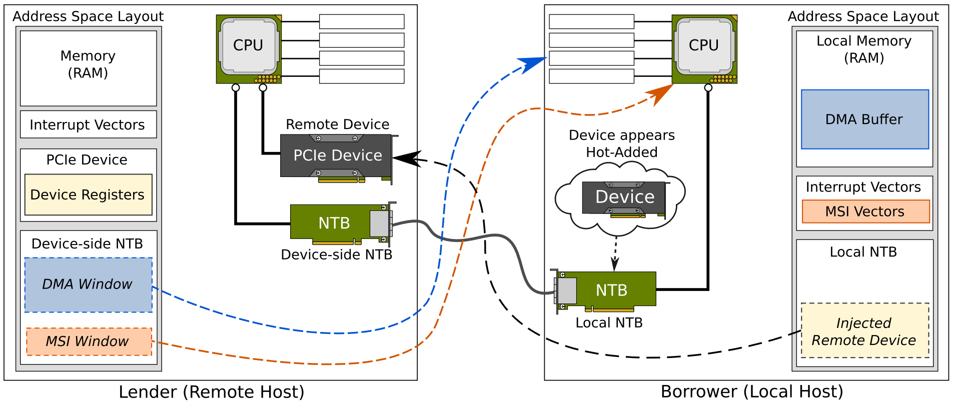


Fig. 8. Device Lending: Using NTBs, it is possible to map the memory regions of a remote device so a local CPU can access device registers. The remote system can in turn reverse-map local resources for the device, making DMA and MSI possible. Device Lending injects a hot-added "shadow device" into the Linux kernel device tree using these mappings, making remote device access transparent to both CPU and device.

4 DEVICE LENDING

By using an NTB, it is possible to map the device memory regions, or *BARs*, of a remote PCIe device (see Figure 8). A local CPU can perform memory operations on a remote device, such as reading from or writing to device registers. Conversely, it is also possible to map local resources for a remote device, allowing it to access memory across the NTB. By making such mappings over the NTB transparent to a device and its driver, it is possible to facilitate use of a device without the system being aware that the device is actually remote. These mappings can be set up dynamically while systems are running, making it possible to reassign devices to different systems without rebooting.

Using this method, we have implemented Device Lending for an unmodified Linux kernel [41]. As illustrated in Figure 8, the implementation is composed of two parts, namely, a "lender," allowing a remote system to use its device, and the "borrower" using the device. In this section, we will describe how we have implemented our Device Lending mechanism.

4.1 Shadow Device

In the Linux kernel, PCIe devices are represented with generic descriptors, providing device drivers with a generic handle that corresponds to a device. This allows Linux to provide a unified interface for functionality that is common for all PCIe devices, such as accessing a device's configuration space, setting up interrupt vectors, memory-mapping device memory and mapping buffers for device DMA. When Linux boots, it enumerates the PCIe device tree as explained in Section 3.2, and generates a corresponding tree of device descriptors.

However, it is possible to manipulate this descriptor tree in software, while the system is running. By implementing our borrower component as part of the NTB driver, we can inject a virtual device, or "shadow device," that appears as an endpoint alongside the NTB for each borrowed device. To Linux, it appears that a (virtual) device has been hot-added [67] to the local system, and it will load any appropriate device drivers using our shadow device as the device handle. In other words, the shadow device acts as a local handle to the remote, borrowed device. By mapping the remote device's memory regions through the local NTB and overriding the shadow device's device memory regions with these mappings, a local device driver may read and write directly to physical device registers without being aware that the device is actually remote.

4.2 Intercepting Configuration Cycles

In order for a device to become aware of the memory addresses used for MSI interrupts, as explained in Section 3.2, the kernel must write these addresses to the device's configuration space. By setting the configuration space accessor functions on our shadow device, we can forward configuration space operations on the shadow device to the remote device in a manner that is transparent to the device driver. However, such interrupts must be mapped over the NTB to trigger the correct interrupt routine on the borrower.

As illustrated in Figure 8, we can prepare a mapping on the device-side NTB to the local interrupt vector assigned by the kernel ("MSI window"). By using the configuration space accessor functions, we can intercept specific configuration cycles and look for writes to the MSI offset, injecting the device-side address of the MSI window mapping into the actual configuration space of the device. This allows interrupts raised by the device to be routed across the NTB and trigger the correct interrupt routines on the borrowing system, transparent to both device and its driver. Additionally, intercepting configuration cycles also makes it possible to mask certain features for the borrower. For example, we can mask legacy interrupts, which can not be mapped over the NTB, so that the device driver will not attempt to use them.

4.3 DMA Window

In order for a device to access local resources using DMA, the lender must set up mappings through the *device-side* NTB to local memory as illustrated in Figure 8. However, it is generally not possible to know in advance which memory addresses a device driver might use for DMA transfers. The pages used for DMA memory buffers may be scattered in physical memory, or an application or device driver may initiate multiple transfers to different parts of memory. Dynamically setting up mappings is not a feasible solution as it would require communication with the lender host and introduce a communication overhead. Additionally, as the number of mappings through the NTB is a finite resource, mapping individual memory pages scales rather poorly.

A naive solution is to make the lender to map the entire physical memory of the borrowing system through the NTB. However, while this would make it possible to set up a single mapping to the remote borrower, the address range of the NTB is not necessarily large enough, as mentioned in Section 3.3; the window on the device-side NTB must be equal to (or larger) than the size of physical memory on a borrowing system to cover the borrower's entire RAM. Moreover, a lender with multiple connected borrowers must potentially map all physical memory of every one of them. In other words, the naive solution would severely limit the number of borrowers as device memory requirements of the NTB itself would become too large.

Modern processor architectures implement an IOMMU, such as Intel's VT-d [3]. The defining feature of the IOMMU is the ability to remap DMA operations issued by a device [38], effectively translating virtual I/O addresses to physical addresses. By using an IOMMU on the borrowing systems, it is possible to remap scattered memory pages to a continuous range. Figure 9 shows how we use the IOMMU on the borrower, allowing the lender to set up a single mapping through the NTB in advance ("DMA window"). When the device driver calls the Linux DMA API to create or map DMA buffers using the shadow device, we inject the device-side address of the DMA window with the appropriate offset, and set up a local IOMMU mapping to the local memory specified by the driver. The device driver passes our injected address to the device, completely unaware that the address is actually a far-side I/O address. This allows the device to reach across the NTB, transparent to both device and device driver.

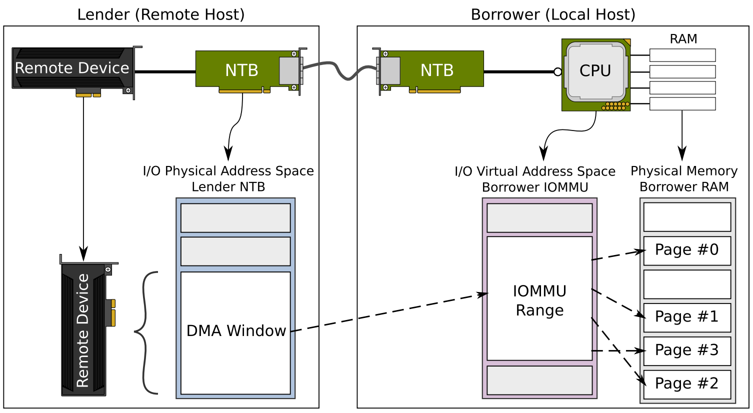


Fig. 9. DMA window: We use the local IOMMU in order create a single continuous memory range. This allows us to conserve NTB resources by setting up a single mapping through the device-side NTB in order for the remote device to reach local RAM. Adding and removing memory pages from the local IOMMU group is inexpensive compared to actively communicating with the lender to set up mappings dynamically.

While our solution adds additional software when a device driver sets up DMA buffers, dynamically adding and removing memory pages from a local IOMMU group has a relatively low overhead compared to communicating with a remote host. Moreover, since mapping across the NTB is done in advance, and all address translations between the different address domains are done in the NTB and IOMMU hardware, our implementation achieves native PCIe performance in the data path.

Some PCIe devices, such as Nvidia GPUs, may have addressing limitations that make them unable to reach higher addresses of the 64-bit I/O address space. For such devices, it can be difficult to configure large enough DMA windows, since the combined memory requirements of the DMA windows must fit through the NTB BAR. Depending on the device memory requirements of downstream devices in the PCIe tree, configuring the NTB BAR size too large may force the system to place the NTB at a high address (see Section 3.1). Because of this, our implementation also supports optionally using the IOMMU on the *lender*. By using the lender's IOMMU, we can remap NTB mappings from high to low addresses if it is necessary, similar to how the IOMMU can be used to avoid so-called "bounce buffering" [52]. An additional benefit is that it also becomes possible to put borrowed devices in their own IOMMU address domains, isolated from other devices in the system. This protects the lender system from any accidental address misconfiguration.

4.4 Shortest Path Routing

Some processing tasks may require the use of multiple devices, such as machine learning workloads that need several GPUs. Such workloads often transfer data from one device to another using DMA, where a device reads from or writes to the memory regions (BARs) of other devices. As described in Section 3.2, shortest path routing between such devices using *peer-to-peer* is possible based on address ranges.

In the case of Device Lending, however, devices installed in different lender systems use different address space domains. The local I/O address used by one host, i.e., the local address a borrower uses to reach a remote device, is not the same address different host would use to reach the same device. Furthermore, a lender may even use an entirely different NTB to reach the other device than it would for reaching the borrower. In order for a borrowed device to reach another borrowed device, we need a mechanism for resolving I/O addresses between the different domains.

With the 4.9 version of the Linux kernel, functionality for setting up mappings between devices to do peer-to-peer DMA between them was added to the device DMA API. By implementing these functions for our injected shadow device, we are notified when a device driver is mapping the device memory regions of another device, and we can inject our prepared mappings. We have implemented the following method of resolving address domains in Device Lending, in order for a borrowed device (the *source*) to reach another borrowed device (the *target*):

(1) **Same lender:** If the *target* is installed in the same host as the *source*, then setting up a mapping is trivial. If the device-side IOMMU is disabled, then the lender simply returns its local device-side I/O addresses of the BARs of the *target*. If the IOMMU is enabled, then the lender additionally needs to set up IOMMU mappings, and returns the I/O virtual addresses.
(2) **Local device:** If the *target* is a device local to the borrower, i.e., residing within the borrowing host, then the *source*'s lender set up DMA windows to the individual BARs of the *target*, similar to how it has already mapped a DMA window to the borrower's RAM. The lender then returns the local device-side I/O addresses the *source* would use to reach through the NTB to reach the the *target*'s BARs. This works for any device in the borrower, even local devices that are not registered with our system. However, in this case, our only works for setting up mappings for a remote device to a local device. The other way around is not possible unless the local device is registered with our system, as we are unable to intercept calls by the device driver without our virtual device handle (shadow device).
(3) **Different lenders:** If the *target* is a remote device, i.e., residing in a different lender host, then the *source*'s lender creates DMA windows through the appropriate NTB to the *target*'s lender. Note that this NTB may be different to the one used to reach the borrower. We then return the local device-side I/O addresses the *source*'s lender would use to reach through the NTB to the the *target*' BARs.

The borrower, after resolving these lender-local I/O addresses, stores them along with its own physical addresses to the BARs of the *target*. When the device driver using the *source* calls the DMA API functions to map the BARs of the *target* for the *source*, the borrower is able to look up the corresponding lender-local I/O addresses and use these. When the driver in turn initiates DMA, it is completely unaware of the location of both the *source* and the *target*, and the *source* will be able to access the *target* through the correct NTB. Figure 10 shows that the the *source* device can reach the *target* device for all three scenarios. By resolving lender-local I/O addresses in advance, we have enabled devices to directly access each other using peer-to-peer. In other words, we have enabled device-to-device communication between remote devices with the lowest possible latency.

5 VM PASS-THROUGH USING MDEV

To provide I/O capabilities to a VM, a VM hypervisor may use emulated devices or paravirtualization. Software-emulated devices appear to the VM guest as an I/O device, but all functionality is handled in the VM implementation. Paravirtualized devices also offer device functionality in software, but relies on facilitation by the hypervisor to use host resources. In many cases, paravirtualized devices are backed by actual hardware. However, emulation and paravirtualization may not be viable options when bare-metal processing power is required.

In this regard, it is possible to to remap DMA and interrupts using an IOMMU. Similarly to pages mapped by an MMU for individual processes, an IOMMU can group 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, enabling direct access to physical memory by the physical device, while other devices and the rest of memory remain isolated. As such, the IOMMU provides a hardware

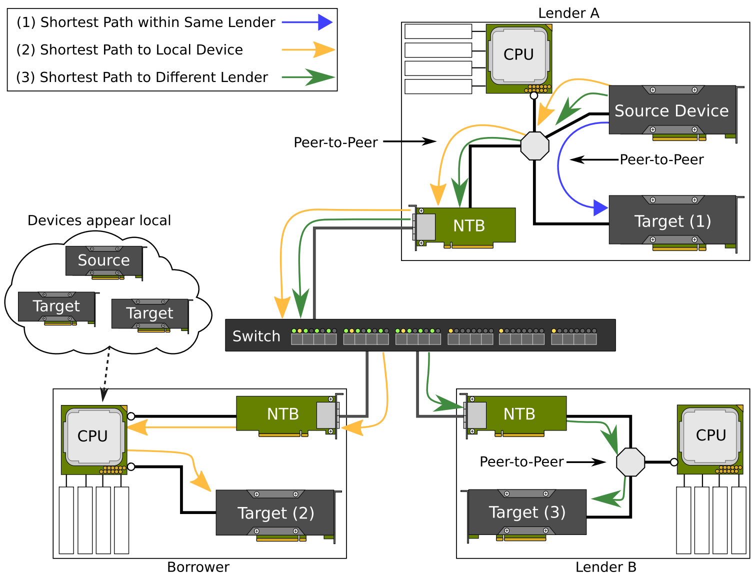


Fig. 10. Shortest path routing: By resolving addresses of device memory regions and preparing mappings for them in advance, we can route device-to-device using the shortest path when a device driver initiates a DMA transfer. Our solution covers all three scenarios: (1) when both devices are in the same lender, (2) when the target device is in the borrower, and (3) when the target device resides in a different lender.

virtualization layer between I/O devices and the rest of the system. This allows a VM hypervisor to facilitate direct access to the physical device from within the VM guest, without compromising the memory isolation provided by the virtualization. This facilitation is often referred to as "pass-through."

In this section, we explain how we have implemented support for such pass-through of *remote* devices in our SmartIO system [48, 49]. We explain how we generalized the core functionality in our Device Lending mechanism, providing us with the necessary software capabilities for implementing a kernel-space interface for the hypervisor. By implementing functionality for dynamically assigning remote devices to VMs, we have extended our device distribution mechanism to support OSes other than Linux, such as Microsoft Windows.

5.1 Mediated Devices

On Linux, pass-through of devices is supported in the KVM hypervisor by using **Virtual Function I/O (VFIO)** [37]. By implementing a VFIO interface for a device, KVM is able to use the IOMMU and map I/O virtual addresses for the device to the same *guest-physical* address layout used by a VM.

Intuitively, a solution for passing through remote devices to a VM would be for the host to borrow a device, injecting the device into its local device tree, and then use VFIO. However, this would not be feasible as VFIO requires that pass-through devices are placed in a separate IOMMU domain per VM guest. As described in Section 4.3, Device Lending places all borrowed devices in the same IOMMU domain to preserve mappings over the NTB. Additionally, pass-through requires the entire guest-physical memory of a VM to be mapped for the device. We need a mechanism for detecting, pinning and mapping the physical memory pages used by the VM instance, in order for the device to be able to DMA to it. VFIO does not provide this mechanism, thus detecting the presence of a VM and mapping its memory is not possible.

In the 4.10 version of the Linux kernel, an extension to VFIO called **mediated device drivers (MDEV)** was introduced [33]. The MDEV extension introduces the concept of a physical parent device having virtual child devices, allowing a host device driver to emulate multiple virtual devices, while still allowing some direct access to hardware. In other words, MDEV facilitates a form of paravirtualization that enables "SR-IOV in software." Some operations on the virtual device, such as configuration cycles and device resets, are trapped (handled) by the parent device driver running on the host, allowing some hardware resources to be emulated while other resources are accessed directly. In our case, using this MDEV interface provides us with a finer-grained control over what the hypervisor and VM guest is attempting to do with the device.

Our implementation registers itself as an MDEV parent device driver for devices under the control of SmartIO. With Device Lending, a device would be exclusively borrowed by the physical host for as long as it runs, regardless of whether any VM instances is using it or not. By implementing functionality for borrowing and releasing device references without injecting them into the local device tree, KVM is able to pass through the device to a VM without it being borrowed first. Only when the VM guest boots up and resets the device, do we actually borrow the device. Similarly, when the guest OS releases the device, either by shutting down or hot-removing the device, we return it. Not only does this limit the lifetime of a borrowed device to when a VM is running and using it, but it also makes it possible to hot-add a device to a live VM.

5.2 Mapping VM Memory for Device

Using Device Lending, we can react to calls to the DMA API on a shadow device to dynamically add or remove pages from the local IOMMU domain. In contrast, we have no way of knowing which addresses a device driver running in the guest may use for DMA. Therefore, the only option is to map all of the guest-physical memory used by the VM for the device.

By using an MDEV parent device driver instead of VFIO, we are aware of a VM instance using the device. However, while the MDEV interface provides us with a method of using KVM to resolve guest-physical addresses to host-physical and pinning the physical memory pages used by the VM instance, we know nothing about the memory layout of a VM instance or even when memory has been allocated. Other implementations using MDEV implement virtual child devices, each with their own set of *emulated* resources. For example, when a guest driver initiates DMA transfers, the parent device driver is notified by trapping emulated device registers, and is able to resolve addresses and pin the appropriate pages in memory just before initiating the DMA engine on the physical device. Our implementation, however, is actually passing through the physical device itself. In our case, the VM instance maps all of the physical device registers and accesses the entire device directly. This 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. This poses a challenge, as the memory used by the VM has not yet been allocated when the virtual device is first picked up by a VM instance.

However, before a PCIe device can use DMA, it must be enabled in a device's configuration space.[4] This allows us to defer mapping of VM memory until our implementation detects a configuration cycle enabling DMA. By then, we can assume that the memory used for the VM is allocated. Even so, we still do not have any information about the address space layout. The naive solution is to map the entire range from start to end. As depicted in Figure 11, this solution is wasteful as a

[4]Enabling the "Bus Master" bit in the command register enables DMA for a device.

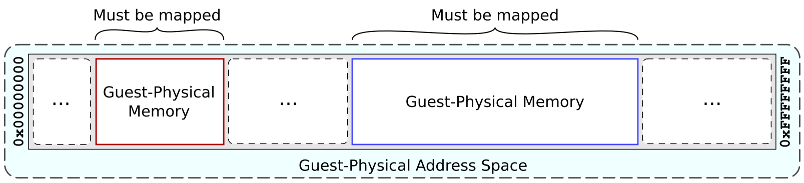


Fig. 11. Mapping VM memory for a device: The VM's address space may be much larger than the actual memory used by the guest. Only guest-physical memory needs to be mapped for a device.

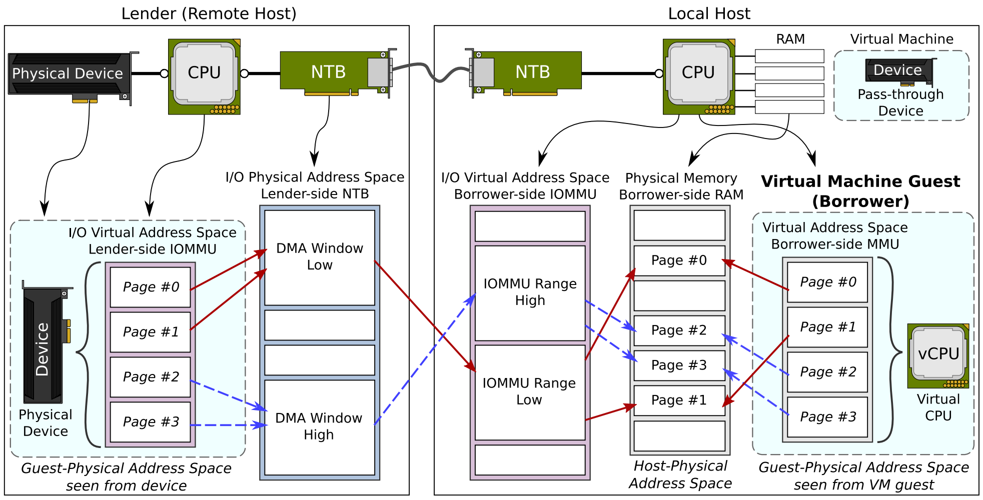


Fig. 12. Pass-through of a remote device: By using IOMMUs on both sides of the NTB, it is possible to map a remote device into a local VM guest's address space. The borrower-side IOMMU provides continuous memory ranges that can be mapped over the NTB, while the lender-side IOMMU is used to map the virtual address space for the device, mirroring the guest-physical layout. We use two windows to map the VM's entire memory.

VM's address space may be much larger than the guest-physical memory size, and not all of this address space should be reachable by the device.

Instead, we can rely on an assumption: as the x86 architecture uses well-defined starting addresses for low and high memory, we can start at these guest-physical addresses and use KVM to experimentally probe which address ranges resolves and which do not. This way, we are able to both dynamically discover the memory layout of the VM and only map those ranges that should be reachable by the device.

Figure 12 illustrates how a device is mapped into the address space of a VM. On the lender, we use the IOMMU to create a virtual I/O address space that maps over the NTB, mirroring the guest-physical memory layout. Because this mapping exists, a native device driver running in the VM guest can initiate DMA transfers on the physical device using guest-physical addresses. On the borrower, we use the IOMMU to provide continuous address ranges that are trivially mapped over the NTB. Note that we create a separate DMA window for the low and high memory ranges, allowing us to map the entire guest-physical memory, while being able to fit through the NTB window.

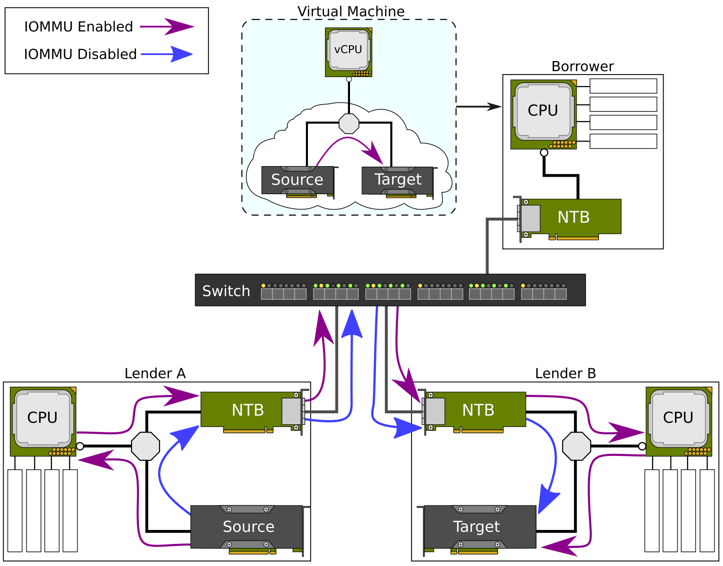


Fig. 13. Since IOMMUs introduce a virtual address space for devices, peer-to-peer transfers must be routed through the root in order for the IOMMU to resolve virtual addresses to physical addresses. As a consequence, shortest path routing is disrupted.

5.3 Peer-to-peer between Devices

Similarly to how guest-physical memory is mapped for a device, the guest-physical BARs of other devices passed through to the same VM can also be mapped for a device. When the guest OS enumerates its PCIe tree and write guest-physical addresses to a device's configuration space, our MDEV parent driver captures these addresses. For all *other* devices, we are able to set up I/O virtual addresses that correspond to these guest-physical addresses using their lenders' IOMMUs. Using the same method described in Section 4.4, we are able to resolve which NTB adapter to map over in order reach the device. This makes it possible to set up mappings between two or more devices using our MDEV implementation, even when they reside in different hosts.

However, while this enables device-to-device access between the physical devices, shortest path routing in the fabric is disrupted by the virtual address space. PCIe transactions must be routed to the IOMMU to resolve I/O virtual addresses to physical addresses (Figure 13). PCI-SIG has developed an extension to the transaction layer that allows devices that have an understanding of I/O virtual addresses to cache resolved addresses called **Address Translation Service (ATS)** [60]. However, ATS is not widely available as it requires hardware support in devices.

5.4 Relaying Interrupts

Similarly to VFIO pass-through, MDEV uses the *eventfd* API [36] to trigger interrupts in a VM instance. When our MDEV parent device driver gets notified to set up an interrupt for a VM, we register an interrupt request handler on the lender for the specified interrupt. Whenever the device raises an interrupt, this interrupt request handler is invoked, which in turn notifies our MDEV implementation. We can then use *eventfd* to signal that an interrupt has been raised to the VM instance.

This method is not ideal, as the latency between a device raising an interrupt and the interrupt routine being invoked within the VM increases. A latency reducing improvement would be to use the same approach as bare-metal Device Lending, and map MSI and MSI-X interrupts over the NTB. However, a benefit of the current implementation is that it allows us to enable legacy interrupts for devices borrowed by a VM, something that is not supported for bare-metal machines.

5.5 VM Migration

As our SmartIO system abstracts away device location, our MDEV implementation supports so-called "cold migration." It is possible to shutdown, migrate, and restart a VM on a different host, while keeping the same passed-through physical devices. If the VM emulator supports it, then it is also possible to hot-add and hot-remove devices to running VMs. Using such hot-swap functionality, live migration could theoretically be possible by first removing all devices, migrating, and then re-attaching them afterwards. However, since such a solution would temporarily disrupt device I/O and force guest drivers to reset all devices, its usefulness would be limited.

Supporting real hot-migration, remapping devices while they are in use without (or with minimal) disruption, is something we wish to implement in future work. Not only would such a solution require keeping memory consistent during the migration warm-up, but DMA transactions could potentially be in-flight during the migration. A mechanism for re-routing transactions, without violating the strict ordering required by PCIe, must be implemented, and will most likely require hardware support that does not exist today.

6 DISTRIBUTED NVME DRIVER

By borrowing a device and inserting it into the local device tree, using either Device Lending or passing the device through to a VM using our MDEV implementation, a device driver may use a device as if it was locally installed. No adaptations are required to use the device, allowing device drivers, OS, and application software to use the device without any modifications.

However, most PCIe device drivers are written in a way that assumes exclusive control over the device. Consequently, a device may only be distributed to a single host at the time, preventing others from accessing it while it is used. This can lead to poor utilization of device resources, as it requires hosts to cooperatively time share a device, resetting it every time it is reassigned to a new host. Some devices implement SR-IOV [62], making a single physical device to appear as multiple virtual devices, allowing each virtual device to be distributed by Device Lending. Regardless, as SR-IOV capability increases the complexity of hardware implementations, it is not widely available, especially for low- to medium-end devices.

During the development of our MDEV implementation (Section 5), we isolated functionality shared with Device Lending and were able to expose this to userspace applications. Effectively, this makes it possible to write device drivers that enable simultaneous sharing and parallel operation of single-function devices by distributing it to multiple hosts at the same time.

In this section, we present our proof-of-concept NVMe driver allowing sharing to multiple hosts simultaneously. NVMe [55] is an interface specification for non-volatile storage controllers that are attached to the PCIe bus, such as **solid state flash memory drives (SSDs)**. Compared to traditional spinning hard disks, where seek time and mechanical disk rotation cause significant delay, these storage drives have lower latency and support parallel operations. This is reflected in the design of NVMe, which supports this parallelism through the use of multiple I/O queues that operate independently and avoiding any form of locking in the I/O submission path. By distributing individual I/O queues, we demonstrate how a single NVMe storage drive may be shared among multiple hosts and operated in parallel.

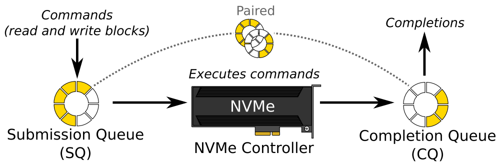
6.1 Device Driver API

We have extended the SISCI API [22] with device-oriented semantics, exposing core SmartIO capabilities through the same shared-memory API used to write cluster applications. In other words, by exposing this functionality through the SISCI API, it becomes possible to implement device drivers as part of the application. Integrating device operation into the application itself makes devices and drivers become part of the same shared global address space as distributed shared-memory applications.

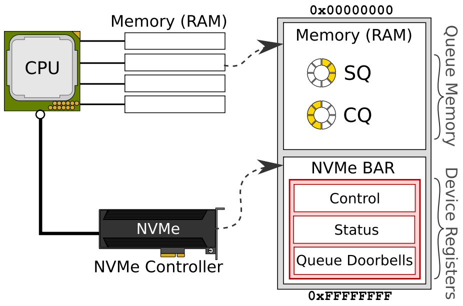
As mentioned in Section 3.3, a userspace application may map "segments" of a remote system's memory into its own virtual address space using SISCI. Moreover, as we explained in Section 4.3, we can set up mappings to such shared memory segments for a *device* as well ("DMA windows"). Devices may use DMA to access shared-memory segments directly, without requiring RDMA. Similarly, by exporting device BARs as shared memory segments, device memory regions may be mapped by several nodes, effectively disaggregating device memory. Memory segments (both system memory and device memory) are associated with *devices*, rather than with hosts. By providing functionality for translating device-side physical addresses, as well as resolving the path through the network between the device and shared memory segments, our API extension allows device driver implementations to be agnostic about address spaces in different cluster nodes. As such, these mechanisms alleviate some of the complexity of implementing distributed device drivers, as software can be written in a way that does not need to consider whether resources are local or remote. The same driver software can run on any node in the cluster, using any device in the cluster, without requiring that the application is actually aware of the specific PCIe topology.

Specifically, the following functionality was added to SISCI:

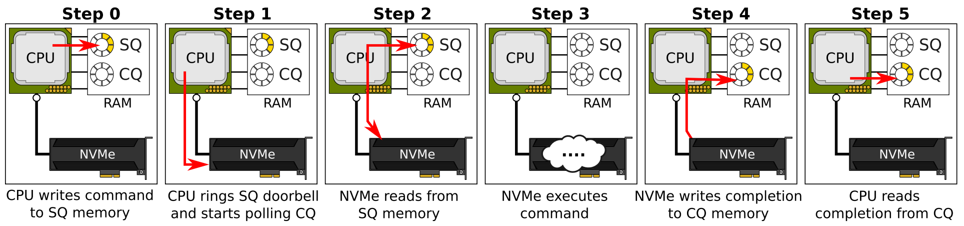
- API functions for letting application processes borrow and return devices. Borrowing a device can either be exclusive, allowing only one borrower at the time, or non-exclusive, allowing several borrowers simultaneously. It possible for a single application process to first take an exclusive reference, to reset, initiate and prepare the device, before allowing other processes in the cluster to borrow the device.
- Automatically exporting device memory regions (device BARs) as segments, allowing them to be memory-mapped into the application process' virtual address space. Additionally, by exporting BARs as segments, it is possible to map them for other devices and set up shortest-path routing.
- API functions for mapping SISCI segments on behalf of a device, effectively setting up DMA windows over the device-side NTB (lender's NTB). This allows the device to use native DMA to read and write to shared memory segments. Segments can be either local or remote to the device, and SmartIO will automatically resolve device-side physical addresses to (remote) memory segments under the hood, allowing the same software to run on any cluster node and remain agnostic about the specific address space layout in other hosts. Note that since BARs of any device registered with SmartIO are automatically exported as SISCI segments, we can map them for other devices as well.
- API functions for allocating SISCI segments using access pattern hinting. While the original SISCI implementation only allows hosts to allocate segments in local system memory, we have added functionality for letting SmartIO choose which host to allocate memory in based on expected access patterns. By relying on hinting rather than actively specifying which host to allocate memory in, we can consider memory locality without requiring awareness of the physical PCIe topology. Note that as these segments are associated with a device rather than cluster nodes, we retain the logical decoupling of machines and devices provided by SmartIO.



(a) NVMe supports asynchronous operation by using a system of paired submission and completion queues.



(b) Queues are allocated in physical memory by device driver software.


Step 0
Step 1
Step 2
Step 3
Step 4
Step 5
CPU
SQ
CQ
RAM
NVMe
CPU writes command to SQ memory
CPU rings SQ doorbell and starts polling CQ
NVMe reads from SQ memory
NVMe executes command
NVMe writes completion to CQ memory
CPU reads completion from CQ

(c) NVMe device operation (left to right). By using DMA, the NVMe device can fetch commands and post completions to queues in system memory.

Fig. 14. NVMe avoids contention in the command submission and completion path by using parallel queues that can be hosted anywhere in physical memory.

Perhaps the most obvious trade-off from using our API extension is that it requires implementing a new device driver. Usually, implementing a driver from scratch entails a considerable engineering effort, and may not even be a viable option in most cases. After all, the main strength of both our Device Lending mechanism and MDEV extension is that they do not require any modifications of existing device drivers. However, as using this API extension allows a device driver to be implemented as part of cluster applications, it is potentially extremely useful for some application domains. By implementing a driver using our API extension, devices can be disaggregated at the software level, rather than at the PCIe device function level. Multiple application processes, running on different nodes, may share devices that do not support SR-IOV. Moreover, not only does our API extension provide an interface for distributed device drivers, but it also becomes possible to write device drivers that fully utilize PCIe shared-memory capabilities. For example, applications may use PCIe multicasting to stream data to several destinations in a single operation. It is even possible to exploit memory locality to optimize data flow through the network.

6.2 Driver Implementation

By avoiding contention in command submission and completion paths and supporting up to 65,535 I/O queues per device, the NVMe standard [55] enables highly parallel operation. Figure 14(a) illustrates how NVMe utilizes a submission and completion queue mechanism. One or more **submission queues (SQs)** are paired with a **completion queue (CQ)**, i.e., multiple SQs may be paired with the same CQ. Commands posted to an SQ will be completed by an entry in the associated CQ. Queues are implemented as ring-buffers, and are allocated in memory by the device driver software as depicted in Figure 14(b). Each queue has its own dedicated doorbell register, avoiding any contention. By allowing queues to operate in parallel, NVMe completely avoids locking and other forms of synchronization between queues.

Figure 14(c) illustrates the basic operation of an NVMe device: The driver software places a command, e.g., "read *N* blocks," into an SQ. It then "rings" the associated SQ doorbell (by writing the SQ tail pointer value). This notifies the NVMe device of how many new commands are ready to be processed. The device fetches commands from SQ memory using DMA. After executing the command, the drive writes a completion to the paired CQ, indicating the status of the operation. The driver must poll CQ memory for new completions,[5] and, as commands may be executed out of order, the driver must keep track of command sequence numbers. Once completions are processed, the driver notifies the NVMe device by updating the CQ doorbell (writing the CQ head pointer value).

To configure I/O queues and manage the device, driver software must first "reset" the device. This is done by clearing a control register on the NVMe controller and writing the base address of the so-called "admin queues," consisting of an admin SQ and an admin CQ. Whereas regular I/O queues use an I/O command set, i.e., reading and writing blocks, the admin queues use a different set of commands for managing the controller, e.g., creating and deleting I/O queues and retrieving controller status.

Our driver implementation consists of a "manager" and one or more "clients," running as userspace software applications. The manager is responsible for initializing the NVMe device, configuring admin queues and relaying admin commands on behalf of clients. A client is a userspace process using one or more I/O queue pairs to read or write data from the NVMe device directly; through using the SISCI API extension described in Section 6.1, the device can DMA directly to application memory with minimal latency. Note that the device manager and clients in this instance are *not* synonymous with the lender and borrowers. Any node in the cluster may run a manager driver, and the same node may even run both a manager driver and client drivers.

Figure 15 illustrates how the driver implementation works. The manager, in this case running on Borrower B, takes control over the NVMe device by using our SISCI SmartIO API extension and borrowing the device. The device registers (NVMe BAR) are already exported as a memory segment, allowing the manager to memory-map them into application address space. Also using the API, the manager allocates a memory segment and maps it for the device (Segment B), retrieving the device-local I/O address (the address, as seen from the device). Finally, the manager resets the NVMe device and sets up admin queues using the device-local I/O address with the appropriate offsets. By having memory-mapped device registers, the manager may "ring" the queue doorbell registers, notifying the device that an admin queue event has occurred. Similarly, as the local memory segment is mapped for the device, the NVMe device is able to fetch commands and post completions over the NTB.

A client driver also borrows the device using the API and memory-maps device registers. Additionally, it can allocate a local segment and map it for the device, retrieving the device-local I/O address. By relaying admin commands using the manager, it can create SQs and CQs using the device-local I/O address. As seen in Figure 15, Borrower A and Borrower C run client drivers and have successfully requested I/O queues for themselves. With these in place, the NVMe device may now be used for I/O, by multiple hosts in parallel. From the point of view of the device, the queues are accessed just like they would be in local memory. In other words, our distributed driver implementation facilitate queue-level sharing of a non-SR-IOV NVMe device, enabling distributed I/O with extremely low latency overhead.

[5]NVMe also supports using MSI/MSI-X interrupts to indicate CQ events, but our implementation relies on completion polling alone.

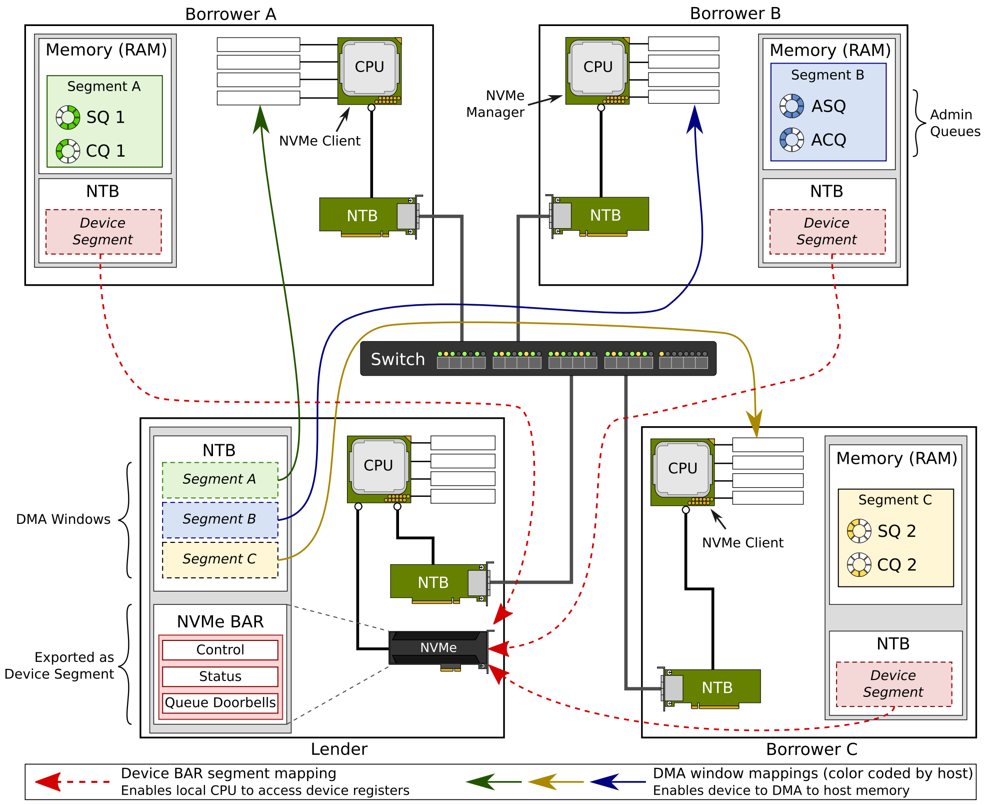


Fig. 15. Simultaneous sharing: The NVMe device can access queues residing in memory segments on different hosts by mapping the segments for the device (DMA windows). Likewise, the borrowers must in turn map the doorbell registers for their respective queues to notify the device about queue events. Each queue has a dedicated register, avoiding any contention between borrowers.

6.3 Multipath Failover

An added benefit of using our SmartIO API extension is that it becomes possible for systems with multiple NTB adapters to borrow the same device through different paths. In the case of our proof-of-concept NVMe driver explained in Section 6.2, it becomes possible to set up redundant I/O queues in advance, and set up mappings through different paths. If the primary path fails, then the driver software may switch over to a backup queue.

Figure 16 illustrates how this is possible: the borrower maps the NVMe device BAR through both its NTB adapters, providing it with two separate paths to the NVMe queue doorbell registers. It can then set up two separate queue pairs in local memory, and by specifying which of the local adapters it is using to reach the NVMe device, our SmartIO system will automatically resolve which of the lender-side NTB adapters to configure DMA windows through. Having established two separate paths, our NVMe driver then chooses one path as its primary path and the other for backup. In the case of a link failure, our NVMe driver is notified either by NVMe I/O command time-out events, or by the low-level NTB driver notifying the NVMe driver about a link event affecting its mapped segments. Reads and writes to mappings that are inactive are terminated by

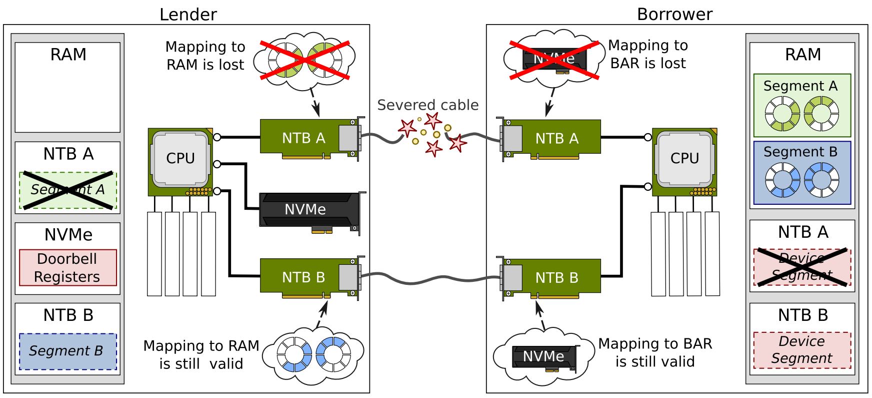


Fig. 16. Multipath failover: We can configure multiple NVMe queue pairs and mapping their memory for the device through different NTB adapters. Similarly, we can also map doorbell registers through separate adapters for the borrower. By having different paths for each I/O queue pair, we can continue operating the NVMe even if one of the paths fail.

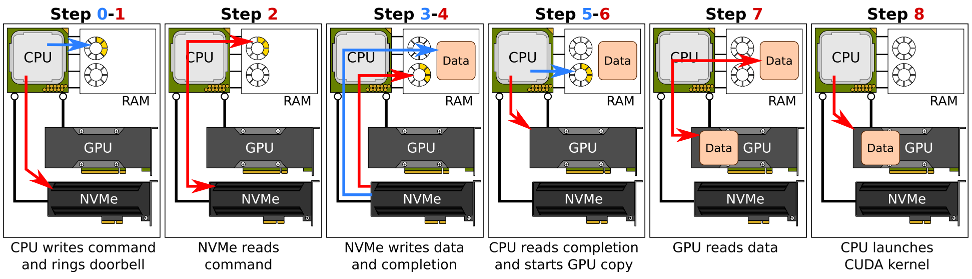
the local NTB adapter.[6] Depending on the kind of failure, for example in the case of a cable being yanked out and plugged back in again, the link may come back up again with mappings still valid. In this case, our NVMe driver can resume using the old queue pair.

The link may become active again with invalid mappings. In the case of I/O queue pairs, this is inconsequential as the NVMe standard supports deleting and creating I/O queues during operation; we can simply delete the old queues, set up new DMA windows and create new queues. However, special care must be taken with regards to the admin queues as they cannot be deleted and recreated without resetting the device and halting all operation. Because of this, we prefer running the manager driver (owning the admin queues) on the lender node. Even if a client's path to the manager is lost, it can have a backup communication path or can re-establish communication if the path comes back up again, without requiring a reset of the device.

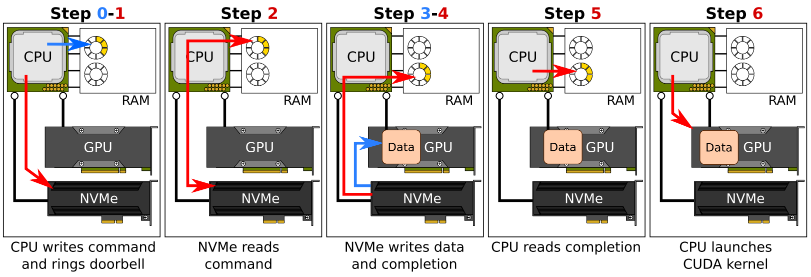
6.4 GPU Support

Many GPU-accelerated applications require fast access to storage. For example, the datasets in big data and machine learning tasks can be hundreds of terabytes. As datasets' size for typical GPU workloads is only increasing, GPU applications become bounded by transfers between storage and GPU. To overcome this, many GPUs permit peer-to-peer DMA to avoid unnecessary copies via system memory [11]. For Nvidia GPUs, such peer-to-peer DMA with third-party devices is supported using GPUDirect [53]. Introduced in the 5.0 version of the CUDA API, memory allocated on the GPU can be exposed through the GPU's device memory regions. This allows third-party devices, such as NVMe devices and network cards, to access GPU memory directly [70, 91]. Figure 17 illustrates the steps involved in reading from storage and loading data onto GPU memory before launching a CUDA kernel[7] on the GPU. The unnecessary steps of first having to read from storage into system memory, and then copying the data to the GPU, as shown in Figure 17(a), can be avoided. Instead, we can map GPU memory for the NVMe (using GPUDirect) and allow the NVMe to access this memory directly using peer-to-peer DMA, as illustrated in Figure 17(b).

[6]Writes are simply dropped by the NTB. Read transactions will result in an unsupported request completion error, which by convention sets all requested bytes to `0xFF`'s.
[7]A software process running on a GPU is called a "kernel" in CUDA. This should not be confused with the OS kernel.



(a) Traditional access: Reading from storage into system memory and then copying it to GPU memory.



(b) Zero-copy read access: Reading from storage directly to GPU memory using peer-to-peer DMA.

Fig. 17. By exposing GPU memory through device memory regions (BARs), it is possible to read from storage directly onto the GPU. This reduces the number of steps required for loading data in to GPU memory.

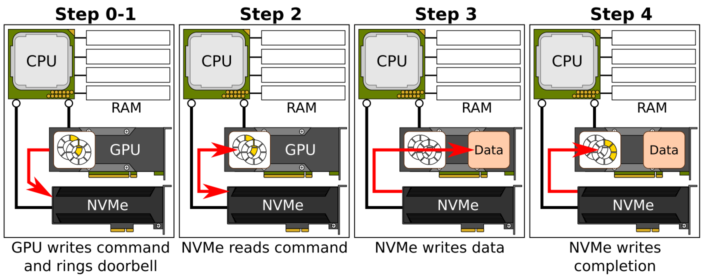


Fig. 18. Avoiding CPU synchronization: By hosting I/O queues in GPU memory and mapping doorbell registers for the GPU, a CUDA kernel running on the GPU can operate the NVMe without involving the CPU.

However, while the CUDA driver does a decent job with regard to pipelining and scheduling, kernel launches are a costly operation from a computational point of view. A better approach would be to avoid interleaving storage I/O and launches altogether, by allowing a long-running kernel to initiate I/O instead. In version 8.0 of CUDA, additional support for registering device memory with the CUDA driver was added to GPUDirect [90]. This feature makes it possible for CUDA applications to use the GPU's onboard DMA engine to access BARs of third-party devices. By memory-mapping the NVMe's BAR, and registering this memory with the CUDA driver, a CUDA kernel can directly access doorbell registers. Similarly, the NVMe is able to fetch commands and post completions to queues that are hosted in GPU memory by exporting GPU memory through GPUDirect. Figure 18 depicts how both features of GPUDirect makes it possible to read from storage directly without involving any software running on the CPU. By operating queues directly, a long-running CUDA kernel can control the NVMe device itself. Loading and storing data can be

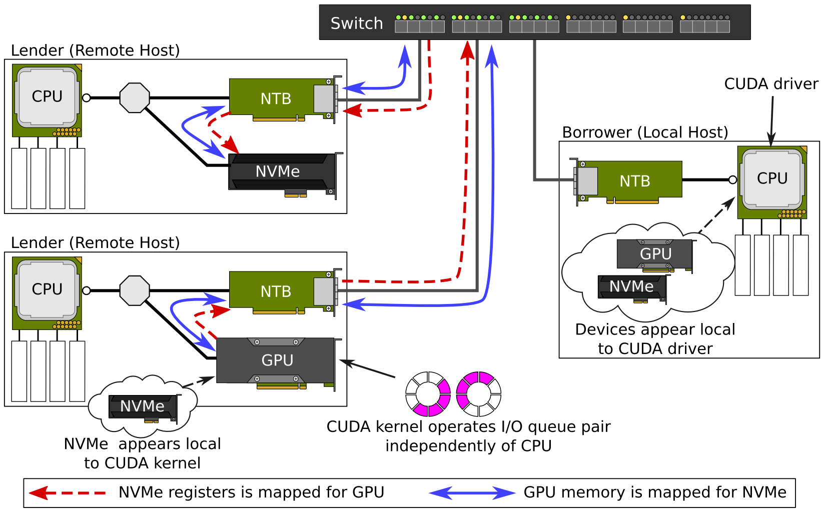

Switch
Lender (Remote Host)
CPU
NTB
NVMe
Lender (Remote Host)
CPU
NTB
GPU
NVMe
NVMe appears local to CUDA kernel
CUDA kernel operates I/O queue pair independently of CPU
CUDA driver
Borrower (Local Host)
NTB
CPU
GPU
NVMe
Devices appear local to CUDA driver
NVMe registers is mapped for GPU
GPU memory is mapped for NVMe

Fig. 19. By combining the SmartIO API extension and Device Lending, our NVMe driver supports direct access to a remote NVMe from a borrowed GPU. To the CUDA driver running on the local system, both the NVMe device and GPU appear local. Our SmartIO system injects necessary peer-to-peer mappings transparently. Note that the GPU operates the NVMe independently; no CPU is needed to access storage.

initiated from the kernel running on the GPU, avoiding the CPU in the data path entirely. Not only does this reduce the latency of loading data onto the GPU, as the kernel may simply batch up read commands and initiate I/O on its own, but we also eliminate needing to schedule data copies from RAM between costly kernel launches.

While controlling an NVMe device directly from a CUDA kernel is interesting in itself, it becomes particularly useful in the context of *remote* devices. Using our SmartIO API extension, our NVMe driver implementation supports CUDA using GPUDirect, allowing queues and data to be accessed directly in GPU memory and "ringing" queue doorbell registers from software on the GPU. As our SmartIO system is aware of device memory regions and their BAR addresses, we can set up such peer-to-peer mappings between remote devices in a manner that is transparent for the CUDA driver. The NVMe may reside in the same host as the GPU, or a different host altogether. Furthermore, the GPU itself may be remote to the host currently running the CUDA driver, as depicted in Figure 19. By using Device Lending and inserting the borrowed GPU into the local device tree, the CUDA application can launch kernels on a remote GPU. Since SmartIO resolves addresses between the different address spaces, the proprietary CUDA driver is completely unaware that both NVMe and GPU are remote devices. To the application, and the local CUDA driver, device memory is available through virtual address pointers that is mapped by our API extension, which are again passed to the GPU when the kernel is launched. This allows the kernel to operate the (remote) NVMe device entirely independent, without involving CPUs or system RAM in the data path at all.

6.5 Multicast

Some NTB-capable switch chips also support multicasting, as described in Section 3.2. Memory writes to a multicast address are routed out on several switch ports. By reserving a continuous address range and dividing it into equal sized "multicast groups," the system can assign different

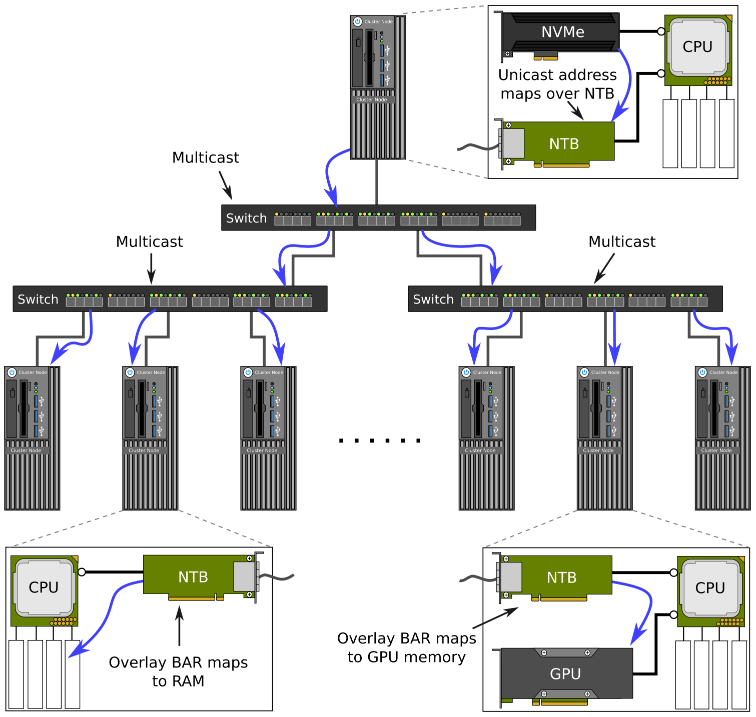


Fig. 20. Multicast support makes it possible for a single DMA operation to be replicated to multiple destinations. It is possible to map multicast destination to system memory and device memory alike.

groups to different switch ports. Subsequently, it is possible to use different destinations for different multicast groups.

However, not all devices support multicast. To overcome this, switches may use a "multicast overlay BAR." If a multicast write matches the overlay BAR on an outgoing switch port, then the top part of the address is replaced with an overlay address. As such, the overlay BAR provides a window into unicast address space for devices (endpoints) that do not support multicast natively. For example, a multicast address may be mapped onto the BAR of a downstream device.

Figure 20 illustrates how we can use multicast to load data from storage to multiple destinations in a single operation. Our SmartIO API extension allows setting up NTB mappings to multicast addresses, allowing a single DMA write operation to be replicated by the switch chip hardware in the cluster switches. When the multicast write reaches the egress NTB adapter, we use an overlay BAR to map the address into anywhere in local address space as long as the destination memory is linear. This makes it possible to set up mappings to either system memory or the BAR of a device, for example GPU memory.

7 PERFORMANCE EVALUATION

The SmartIO system makes it possible to distribute PCIe devices in a PCIe-interconnected cluster. Our implementation relies on several software and hardware components that enable access to remote devices over the network. We have evaluated Device Lending and the MDEV extension in our previous work, explaining performance differences as being caused by increased latency from longer PCIe paths [48, 49]. However, by setting up the necessary memory-mappings in advance and injecting these prepared mappings during use of the device as explained in Section 4.3, there should not be *any* impact on performance. After all, we only rely on native PCIe in the critical path. Although it may be extrapolated from our previous results that Device Lending and MDEV does not cause any performance degradation, it is not concrete evidence. The assertion that our SmartIO system has no performance overhead compared to local access warrants proper investigation, something our previous evaluations partly lacked.

To remedy this, we present here an evaluation consisting of several, entirely new performance experiments. These new experiments are designed to verify that our sharing techniques themselves do not add any performance penalty compared to local access. By comparing the performance of using remote devices to using devices attached to a local PCIe bus, thus establishing a "local baseline" for comparison, any overhead caused by our implementation should be revealed. All parts comprising our SmartIO system is evaluated from multiple angles to verify that our solution is in fact "zero-overhead." Not only do we here revalidate our previous findings [48, 49], but we also argue that this improved evaluation supersedes our previous work, as we present updated performance results using more recent hardware. In addition, we present evaluations on other parts of the system that have not been presented in earlier work, such as an isolated latency analysis of our memory-mapping routines, and an evaluation of the shared-memory capabilities of our new NVMe driver. In total, this gives a complete evaluation of the entire SmartIO system.

We have organized the evaluation of the different components of our SmartIO system as follows:

- In Section 7.1, we perform a series of experiments comparing Device Lending to local configurations, showing that our implementation does not cause any performance degradation beyond what is expected for deeper PCIe device trees. Additionally, we prove the capability of running unmodified software and device drivers, by using standard benchmarking applications and native device drivers.
- In Section 7.2, we evaluate the usefulness of Device Lending for realistic workloads by presenting the performance of an image classification application implemented for Keras and Tensorflow [1, 2]. By training a convolutional neural network using several remote devices from different hosts, we prove the capability of Device Lending for scaling heavy workloads.
- We evaluate our MDEV implementation in Section 7.3, where we pass-through physical GPUs to a VM guest and benchmark DMA performance. We are able to demonstrate that our implementation achieves the same performance as bare-metal configurations.
- Experiments using our distributed NVMe driver are presented in Section 7.4. We demonstrate the flexibility of shared-memory clustering and our distributed device driver API by demonstrating how memory locality can be fully exploited to reduce latency. Additionally, we prove the latency benefit of using PCIe networking by comparing our implementation to a state-of-the-art NVMe-oF implementation using InfiniBand RDMA.

Note that throughout our evaluation, we have used different software versions for the different experiments, such as different Linux distributions and CUDA installations. This is to fully demonstrate that our SmartIO system is not limited to a specific Linux version, but works for a wide variety of distributions and software versions, including older versions. We make a point of using standard and unmodified benchmarking software for our tests. Furthermore, while we relied mostly on GPUs in our previous evaluations, we present here results using GPUs, network adapters, and NVMe devices to show that we can share any standard PCIe device. This has the added benefit of demonstrating several sharing scenarios for a range of applications, which are made possible by our SmartIO solution. For each set of experiments, we explicitly state what kind of hardware and software is used in the configuration.

7.1 Device Lending

Our previous Device Lending evaluations focused on investigating how the increased latency from longer PCIe paths affects performance with regard to increased DMA latency and decreased link utilization [48, 49]. In the past, we have argued that this difference in performance is very small when compared to other device distribution mechanisms, such as RDMA. While it may be extrapolated from our results that our implementation does not cause any performance degradation, it is not sufficient evidence by itself that the performance difference is *only* caused by additional switch chips in the PCIe paths.

Device Lending makes it is possible for application software on a local system to use remote devices without requiring any modifications to device drivers, or even the OS. Comparing the performance of using remote devices to a local baseline can be done by creating local PCIe device trees that are as similar to to the Device Lending scenarios as possible, since all other conditions are the same. We have performed a series of new experiments comparing Device Lending scenarios to local performance using a BP-457-ATX PCIe expansion chassis, to create PCIe paths with the same number of switch chips (or “hops”) for both local and remote topologies.

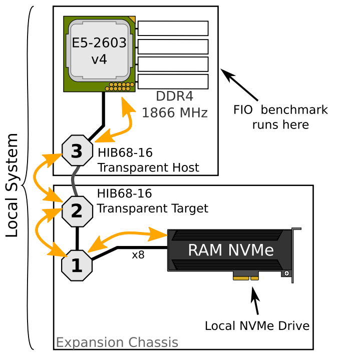
7.1.1 Latency Tests. To prepare a DMA transfer, memory must be mapped for a device. This involves locking pages in memory so they are not swapped out and resolving their I/O addresses. For reading from block device, i.e., a storage device, the Linux block-layer pin the pages used by a memory buffer and create a scatter/gather list containing the physical addresses of the buffer. This list is then passed to the device driver, which in turns iterates the list and resolves I/O addresses by using the Linux DMA API. If the IOMMU is enabled, then the same API is used to set up IOMMU mappings for the device. The driver can then use these I/O addresses and initiate DMA transfers.

As explained in Section 4.3, by inserting a shadow device into the local PCIe tree, our Device Lending mechanism has a “hook” in the DMA API. When the device driver calls the DMA API using the shadow device, we can calculate offsets and inject corresponding I/O addresses that map over the device-side NTB. This allows us to prepare mappings over the NTB in advance (“DMA windows”), and no communication between the lender and borrower is required. However, the software routine that calculates offsets may still have an impact on performance, particularly in the case of device drivers that frequently maps and unmaps memory for a device.

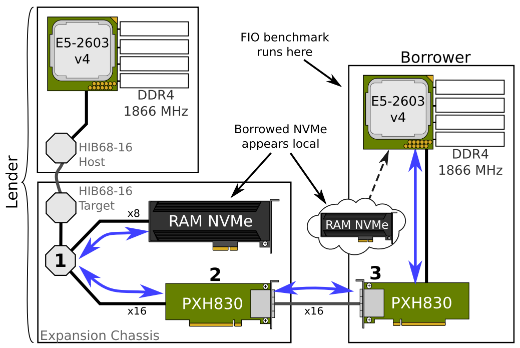
To measure any performance impact of our mapping routine, we have used the ***Flexible I/O tester* (FIO)** [9]. FIO is a widely used userspace application for benchmarking the performance of storage devices, such as NVMe devices. By configuring FIO to perform reads and using the *sync* engine, FIO opens a file descriptor to the block-device setting the `O_DIRECT` and `O_SYNC` options. This combination of options allows Linux to perform zero-copy reads from storage, bypassing the block-cache and forcing the block-layer and NVMe driver to map and unmap the userspace buffer for every single read operation. In other words, this FIO benchmark configuration forces our mapping routine to be invoked as part of the critical path.

Figure 21 shows the hardware topologies for our test scenarios:

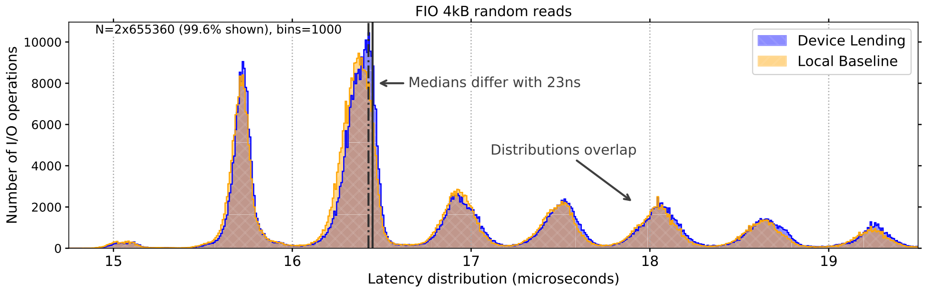
- **Local Baseline**, shown in Figure 21(a): An expansion chassis connected to a local host running CentOS 7, using the 3.10 version of the kernel and the built-in NVMe driver. We are running FIO version 3.7 as available from the CentOS 7 software repositories.
  The expansion chassis is connected to the upstream host through One Stop Systems HIB68-x16 target adapter cards. These adapters use the same Broadcom PEX8733 switch chip used



(a) **Local Baseline**: an NVMe device in an expansion chassis attached to the local PCIe bus.



(b) **Device Lending**: a borrowed remote NVMe device, appearling local to the system. Note that the number of hops in the PCIe path is the same as in the local baseline.



(c) Histogram of the latency distribution for reads from storage for both the local and remote scenarios. Each data point is the latency of a full a 4 kB read operation. The distributions for the Local Baseline and Device Lending overlap, demonstrating that our software implementation does not add any overhead when it is part of the critical path.

Fig. 21. We benchmark our Device Lending driver software by using an NVMe benchmark that calls our mapping code in the critical path (FIO). By using an expansion chassis, the NVMe device is the same number of hops away from the CPU currently using it for both the Local Baseline comparison and Device Lending. The only difference is whether the switch chip is configured in transparent or NTB mode.

in the Dolphin PXH830 NTB adapters.[8] By placing the NVMe device in an expansion chassis, we were able to create a similar PCIe path for both test scenarios. Additonally, the IOMMU was disabled, to make the configuration comparable to Device Lending described below.

- **Device Lending**, shown in Figure 21(b): Two are connected together in a back-to-back topology, using Dolphin PXH830 adapter cards and external PCIe cables Both hosts are running CentOS 7 with the 3.10 kernel, and the local system running the benchmark has borrowed the remote NVMe and inserted it into the local PCIe tree and using the in-kernel NVMe driver. The IOMMU on the borrower is disabled, and we have configured the the DMA window size large enough to map the entire memory of the borrowing system. By disabling the IOMMU on the borrower, we make sure that the only latency overhead is our own mapping routine. The same expansion chassis configuration as in the local baseline is used, and by

[8]While it is possible to configure the PXH830 adapter cards in transparent mode rather than NTB mode, the One Stop Systems expansion chassis used in our tests uses a non-standard connector pin for the PCIe clock signal. In lieu of the possibility of putting the PXH830 in transparent mode, we therefore use HIB68-x16 adapters.

disabling the IOMMU on the lender, PCIe transactions are routed peer-to-peer as illustrated. This ensures that the NVMe device is the same number of switch chips away from the CPU currently using it, making the configuration comparable to the local baseline configuration described above. The only difference is whether the switch chip in the adapter cards is configured in *transparent* or *non-transparent* mode (NTB).[9]

In both scenarios, FIO was configured to perform 8,192 reads per run, each read is a page-sized (4 kB) chunk at an offset generated by a pseudo-random generator. As FIO reuses the same buffer for every read call, we ran FIO several times and concatenated the results. In addition, we reloaded the NVMe driver between each fourth run to force the system to use different memory locations for the internal I/O command queues. Moreover, we also rebooted the system between each eighth run of FIO to ensure that the results were the same across multiple system reboots. In short, for both scenarios, we had 10 reboots, 2 reloads of the NVMe driver per reboot, 4 FIO runs per driver reload, and 8,192 read operations per run. As the purpose of this test is not to benchmark the performance of the NVMe device, but rather a potential overhead of our Device Lending mechanism, the NVMe drive we have used is a prototype RAM disk with an NVMe controller from PMC-Sierra. This is to avoid any effects caused by prefetching and caching that modern SSDs are capable of.

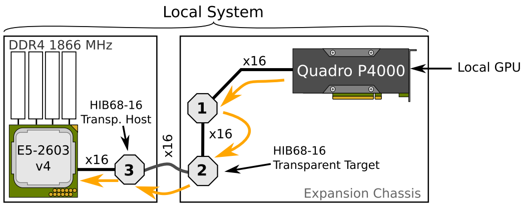
Figure 21(c) shows the latency distribution of read operations for both using a local NVMe device (Local Baseline) and when accessing a remote NVMe device using Device Lending. Although the purpose of the test is simply to compare Device Lending to local access, it is interesting to note that the distributions have distinctive "spikes" occurring at regular intervals. We suspect that these spikes may be caused by a combination of task scheduling in the kernel and interrupt aggregation by the NVMe device. We see that the two distributions overlap, and the medians differ with 23 ns. Considering the spread of the distribution, this is not statistically significant. We argue that this demonstrates that there is no significant difference in performance for local and remote.

7.1.2 Throughput Tests. As mentioned in Section 4, it is not feasible for a lender to map the entire memory of multiple borrowers in a cluster. This would potentially require setting the NTB BAR size larger than what system limitations permits. Furthermore, not all devices support high I/O addresses, and such devices would be unable reach the higher address offsets of the NTB for large DMA windows. To overcome this, our implementation uses the IOMMU on the *borrower* instead. By using the borrower-side IOMMU, we can create continuous address ranges using predetermined I/O addresses. These continuous ranges are trivially mapped by the device-side NTB (DMA windows) and can be done in advance. However, this requires dynamically adding memory pages to the IOMMU domain when the device driver is preparing DMA buffers. Our implementation must also make sure to not choose virtual I/O addresses that risk thrashing the IOMMU translation look-aside buffer [7].

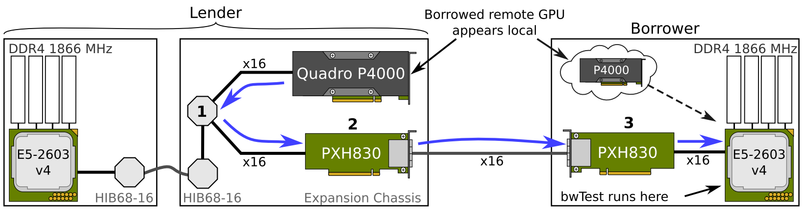
By performing large DMA transfers, we saturate the PCIe links with DMA traffic and also stress system memory. This allows us to investigate if there is any performance difference between using a local device or a borrowed, remote device for high-throughput workloads. Any overhead caused by our IOMMU support would show as a noticeable performance difference in the achieved memory throughput. Figure 22 shows the hardware topologies used in our tests:

- **Local Baseline**, shown in Figure 22(a): A local system using a local Nvidia Quadro P4000 GPU in an expansion chassis. As with the NVMe tests, we use an expansion chassis to make the PCIe path similar to the Device Lending scenario. The IOMMU on the local

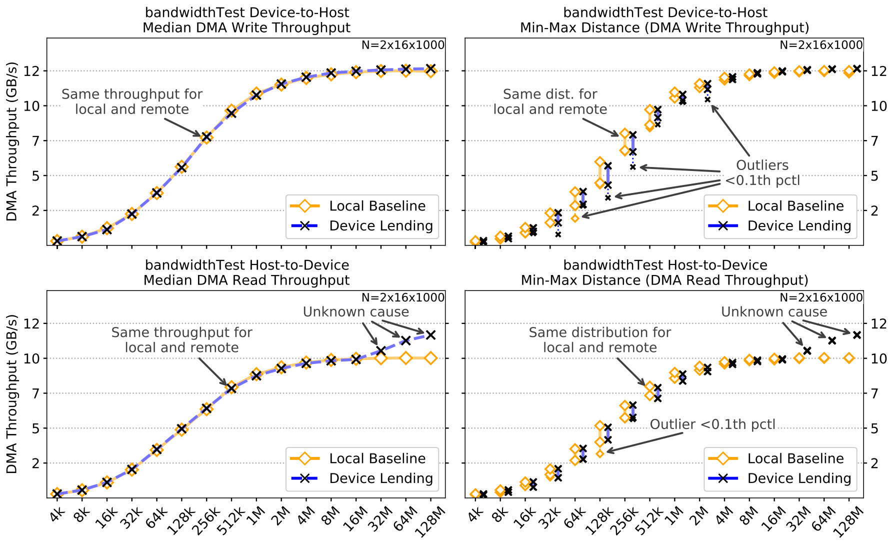
[9]Since an NVMe read operation involves a register write, several DMA transactions and interrupts, comparing similar hardware topologies would also reveal any latency overhead in the address translation mechanism of the NTBs as well.



(a) **Local Baseline**: using a local GPU in an expansion chassis. Note that the transparent host and target adapters use the same PEX8733 switch chip as the Dolphin PXH830 NTB adapter. The route for DMA writes is shown. The IOMMU is enabled in order to make the scenario comparable to Device Lending.



(b) **Device Lending**: using a borrowed remote GPU from a lender connected back-to-back. By disabling the IOMMU on the lender, transactions are routed shortest path. The route for DMA writes is shown. Note that this has the same number of hops in the PCIe path as the Local Baseline. By enabling the IOMMU on the borrower, we can shrink the DMA window size.



(c) The throughput for DMA writes to system memory (top row) and DMA reads from system memory (bottom row). The different transfer sizes are plotted along the X-axis, and for each size we initiated 1000 transfers. Median throughput is shown on the left, and the distribution as min–max distance on the right.

Fig. 22. By performing large DMA transfers, any overhead in the critical path would have been revealed as a difference in performance. As performance is the same for Device Lending and the Local Baseline, this is not the case, and the performance is indeed similar.

CPU is enabled, and the Linux kernel decides IOMMU mappings. This makes the scenario comparable to the Device Lending scenario below.

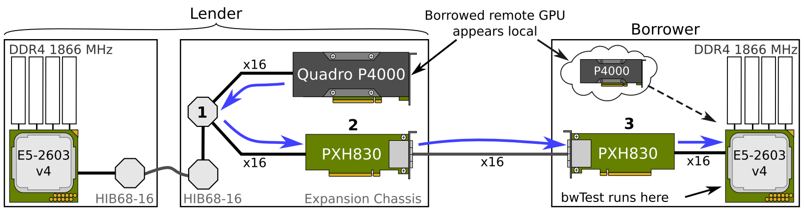
- **Device Lending**, shown in Figure 22(b): Two hosts connected back-to-back using Dolphin PXH830 NTB adapter cards, one host is borrowing the Quadro P4000 GPU. The IOMMU on the lender host is disabled, to enable DMA transfers to be routed shortest path over the NTB in the expansion chassis, making this scenario comparable to the Local Baseline. Since the GPU used in our tests is unable to reach high I/O addresses, mapping the entire memory of the borrower is not possible. Because of this, we configured the NTB BAR size to 1 GB. This is small enough for the system to place the NTB at low addresses during the PCIe bus enumeration described in Section 3.1. Since the IOMMU on the borrower is enabled, we can detect any overhead in how we use the IOMMU compared to the Local Baseline.

We installed version 10.1 of CUDA (418.39 version of the Nvidia driver), and the systems are running Ubuntu 18.04.2 with the 4.15 version of the kernel. We used the *bandwidthTest* program to create the workload. This CUDA program uses the GPU's on-board DMA engine to copy data between GPU memory and system memory, and is included in the CUDA Toolkit sample programs [54]. For both scenarios, we configured bandwidthTest to initiate 1,000 DMA writes to system memory, and 1,000 DMA reads from system memory. We repeated this for sizes from 4 kB to 128 MB, to reveal any trends in increased transfer sizes.

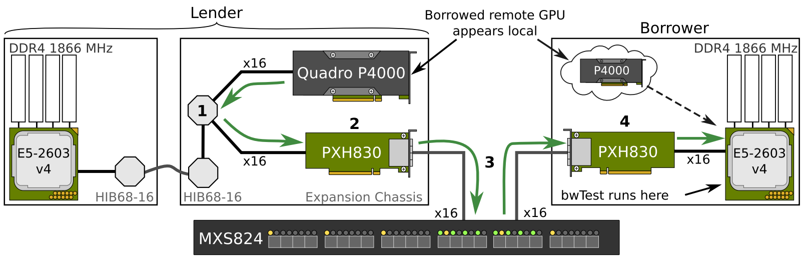
Figure 22(c) depicts the results of our test, with DMA writes in the top row and DMA reads in the bottom row. The different transfer sizes are plotted along the X-axis. The left column depicts the median of 1,000 runs. To show that even the distribution of measurements are similar for local and remote, we depict the min–max distance of the reported throughput samples on the right column. Since the Nvidia driver actively trains down the PCIe link to conserve power consumption, we enabled persistence mode on the GPU. However, this was not enough to entirely avoid that the GPU's DMA engine takes some time to "warm up" caches on the GPU. Because of this, measurements below the 0.1th percentile are marked as outliers. The throughput for Local Baseline and Back-to-Back scenarios overlap almost perfectly, which should be interpreted as a strong indication that our Device Lending implementation does not introduce any overhead compared to local performance. Finally, we also observe a strange effect for DMA reads where the achieved throughput for Device Lending appears to overtake local performance. This "boost" is statistically significant, as can be seen in the min–max plot. We do not fully understand what causes this effect, but we suspect that it may be caused by different IOMMU mappings for the Local Baseline and Device Lending scenarios, since they are decided by the kernel and our implementation, respectively.

7.1.3 Longer PCIe Paths. PCIe transactions are either *posted* or *non-posted* operations, meaning that some transactions require a completion to be sent back. DMA reads are requests that require a completion with data. As such, reads are affected by the number of hops in the data path between requester and completer; the longer the path, the higher the request-completion latency becomes. In addition, the PCIe data link layer uses a credit-based flow control algorithm. The number of requests in flight is limited by how many uncompleted transactions a PCIe requester is able to keep open. Since it is not allowed to send more than the maximum payload size at the time,[10] a requester may need to split requests into several transactions. Longer paths can therefore reduce DMA performance, as the link becomes underutilized when the distance between device and memory increases.

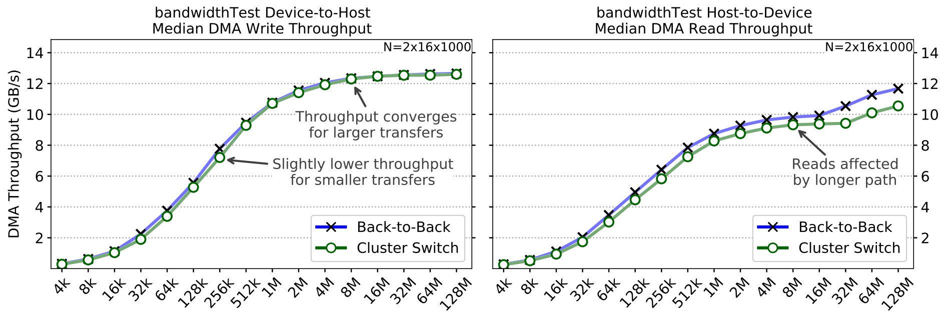
[10]The maximum payload size for a device is configured by the system. While it can be configured individually for each device, it is usually configured to be the same for all devices in the PCIe tree due to several practical reasons, and is most commonly set to 128, 256, or 512 bytes.



(a) **Back-to-Back**: two hosts connected back to back. The path for DMA writes is illustrated.



(b) **Cluster Switch**: two hosts connected via a Dolphin MXS824 cluster switch, increasing the PCIe path by an additional hop compared to the Back-to-Back scenario. The path for DMA writes is illustrated.



(c) The median throughput for DMA writes to system memory (left) and DMA reads from system memory (right) using the bandwidthTest CUDA program. The different transfer sizes are plotted long the X-axis.

Fig. 23. By increasing the distance with a single hop, we are able to determine the impact of longer PCIe paths on DMA performance. DMA reads are particularly affected by the decreased link utilization.

We used the bandwidthTest program described in Section 7.1.2 and a borrowed, remote Nvidia Quadro P4000 GPU. Figure 23 shows the topologies used to evaluate the performance impact of increased PCIe paths. By increasing the distance between device and memory with an additional hop, namely, the Microsemi PM8536 PFX switch used internally in the MSX824 cluster switch, we can compare the performance to the Back-to-Back scenario. The hosts are running Ubuntu 18.02.2 with the 4.15 version of the Linux kernel. As with our previous tests, we used CUDA version 10.1.

Figure 23(c) shows the results of our test. As expected, the additional hop in the Cluster Switch scenario affects DMA performance. We can see that smaller writes are affected by the increased latency through the switch, because even small differences in delay impact the time it takes for transactions with data to arrive. However, this additional latency becomes less significant for larger writes, as the number of transactions in flight increases. We see that the throughput converges towards the Back-to-Back performance for transfers larger than 512 kB.

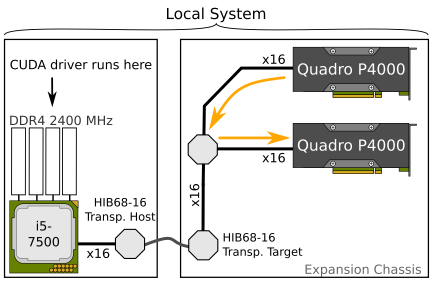
DMA reads suffer noticeably from the increased distance. Unlike writes, which are posted transactions, the number of read requests simultaneously being held open is limited. Moreover, PCIe allows a completer to respond with less data at the time than is actually requested. For example, a read requesting 512 bytes may terminate with 2 completions containing 256 bytes each, rather than a single completion with all 512 bytes. This depends on the maximum payload size and maximum read request size, configured by the system. Since the time before completions arrive increases because of the longer distance between the GPU and system memory, the link becomes underutilized as there are fewer transactions in flight. We observe this as a drop in the measured throughput, as seen on the right-hand plot in Figure 23(c).

7.1.4 Peer-to-peer: Local vs. Remote. In addition to enabling access to individual remote devices, Device Lending also supports creating groups of arbitrary devices and enabling direct peer-to-peer access between them (shortest-path routing). To show that the address resolving method described in Section 4.4 enables shortest-path routing and to demonstrate that relying on the borrower-side IOMMU does not disrupt peer-to-peer transactions on the lender, we have performed DMA throughput and latency tests using two Nvidia Quadro P4000 GPUs. The borrower uses CUDA 10.1 with the 418.39 version of the Nvidia driver, and both borrower and lender run Ubuntu 18.04.2 with the 4.15 version of the Linux kernel. The configurations of the tests are shown in Figure 24:

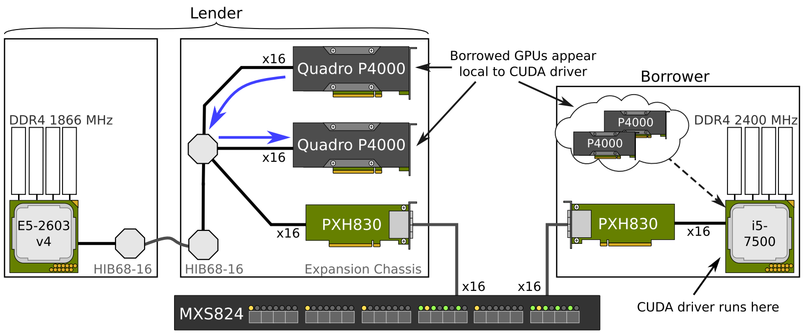
- **Local Baseline**, shown in Figure 24(a): A local system using two local GPUs in an expansion chassis. We have disabled the IOMMU on the local CPU, to enable shortest path routing within the expansion chassis.
- **Device Lending**, shown in Figure 24(b): Two hosts connected together using Dolphin PXH830 NTB adapter cards. Note that we also use a Dolphin MXS824 PCIe cluster switch in this test. Even though the switch increases the distance between CPU and two GPUs, it does not matter in this test; we only measure traffic between the two GPUs. The IOMMU on the lender is disabled to allow shortest path routing. Since the GPUs used in our tests are unable to reach high I/O addresses, we configured the DMA window size to 1 GB and enabled the IOMMU on the borrower.

Figure 24(c) shows the result of using the CUDA bandwidthTest program to copy memory from one GPU to the other using the first GPU's on-board DMA engine. For each transfer size, we configured bandwidthTest to do 1,000 transfers. On the left, we show the median throughput, and we show the distribution as a min–max distance on the right. Note that GPU memory latency varies significantly more than RAM (as seen in Figure 22).

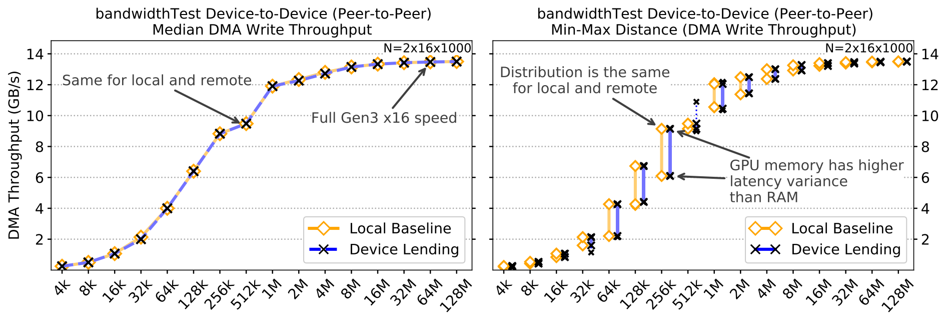
Using the same topologies as depicted in Figure 24, we have also measured the latency of DMA writes between the two GPUs. We developed a small CUDA program to measure peer-to-peer latency, as depicted in Figure 25(a). One GPU is tasked with increasing a counter, writing it to the other GPU's memory and waiting for an acknowledgement. The other GPU waits for the counter to increase by one, and acknowledges the received counter by writing it back to the first GPU. The whole round-trip is measured by recording the current GPU clock cycle and dividing it by the clock frequency. We call the elapsed time of one cycle of DMA transfers back and forth the *ping–pong* latency. For getting the clock cycles, we use the `clock64()` function. We measured that calling this function takes around 32 ns on the P4000 GPUs. We also measured that reading from the local memory pointer takes around 15 ns. While this skews the results somewhat, we argue that the skew should be identical for both scenarios.



(a) **Local Baseline**: using two local GPUs in an expansion chassis. By disabling the IOMMU, DMA writes from one GPU to the other is routed shortest path.



(b) **Device Lending**: using two borrowed remote GPUs from the same lender. By disabling the IOMMU on the lender host, DMA writes are routed peer-to-peer similarly to a local system. The borrower-side IOMMU is enabled in order to avoid mapping the entire memory of the borrowing system. Even though the switch increases the distance between CPU and devices, we are only measuring traffic between the GPUs.



(c) The throughput distribution of DMA writes to a peering GPU. The spread is higher than for system memory because of GPU memory latency. Note that the distributions for the Local Baseline and Device Lending overlap, again indicating that our implementation does not add any overhead in the critical path.

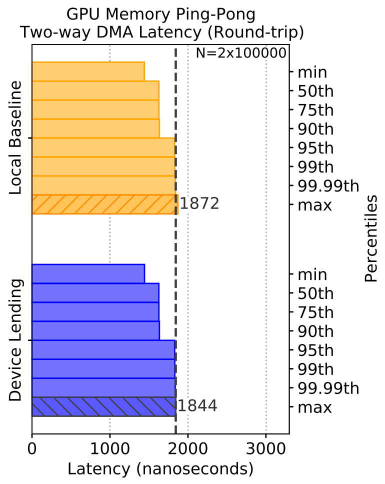
Fig. 24. Peer-to-peer throughput: We demonstrate that our Device Lending implementation supports shortest path routing by comparing peer-to-peer DMA performance. The IOMMU on the borrowing system does not affect traffic between borrowed devices.

```
__global__ void pingKernel(
        volatile uint32_t *remote, // Memory on pong GPU
        volatile uint32_t *local, // Local GPU memory
        uint64_t *times)
{
    for (uint32_t i = 1; i <= N; ++i) {
        uint64_t before = clock64();
        *remote = i; // Send ping
        while (*local < i); // Wait for pong
        uint64_t after = clock64();

        // Record clock cycles
        // we divide this by clock frequency later
        times[i-1] = after - before;
    }
}
```

```
__global__ void pongKernel(
        volatile uint32_t *remote, // Memory on ping GPU
        volatile uint32_t *local) // Local GPU memory
{
    for (uint32_t i = 1; i <= N; ++i) {
        while (*local < i); // Wait for ping
        *remote = i; // Send pong
    }
}
```

(a) The "ping" and "pong" CUDA kernels used in our peer-to-peer latency tests.



(b) Distribution of ping-pong latency measurements. Note that the distributions are similar for local and remote, indicating that our implementation does not add any overhead.

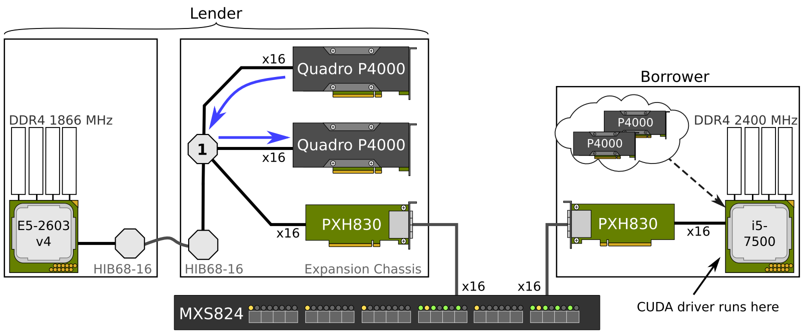
Fig. 25. Peer-to-peer latency: by implementing a ping-pong program in CUDA, we can measure the latency of DMA writes between two GPUs. One GPU writes a 4-byte message to the other GPU's memory, before waiting for an acknowledgement and recording the time before and after (ping). The other GPU waits for the message and sends an acknowledgement back (pong).

Figure 25(b) shows the latency distributions for the Local Baseline and Device Lending scenarios for 100,000 ping-pong iterations each. As the distribution has three distinct "steps," with no measurements falling in between, we present it as a set of percentiles rather than a histogram. We see that the distributions of throughput and latency measurements are similar for both scenarios, proving that there is no difference between local and remote. From this, we can conclude that our implementation supports shortest-path routing between two devices, without adding any overhead in the critical path.

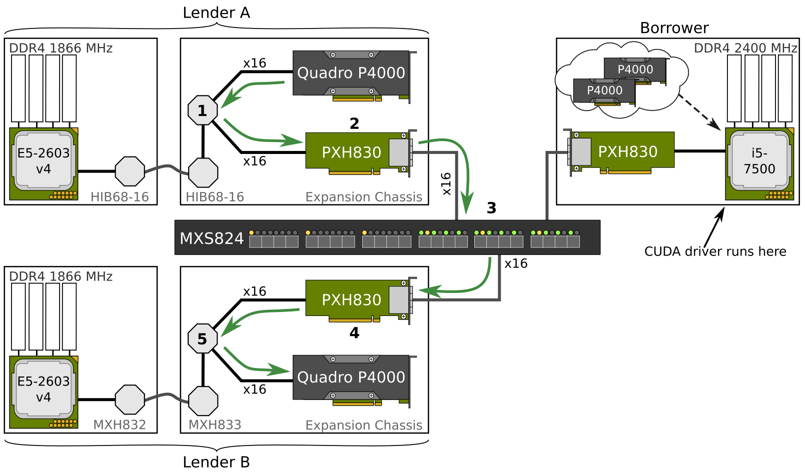
7.1.5 Peer-to-peer: Multiple Lenders. As described in Section 4.4, our Device Lending implementation also supports shortest-path routing between devices even when they reside in *different* lender systems. By composing a PCIe infrastructure consisting of devices spread out over multiple hosts in the cluster, the PCIe device tree unavoidably becomes deeper. While this can potentially increase resource utilization significantly, we need to evaluate the performance impact of moving resources further away as each additional hop in the data path will slightly increase the latency.

By using the same peer-to-peer benchmarks described in the previous section, we have evaluated the impact of moving one of the GPUs to a third host. Figure 26 illustrates the topologies of our comparison tests:

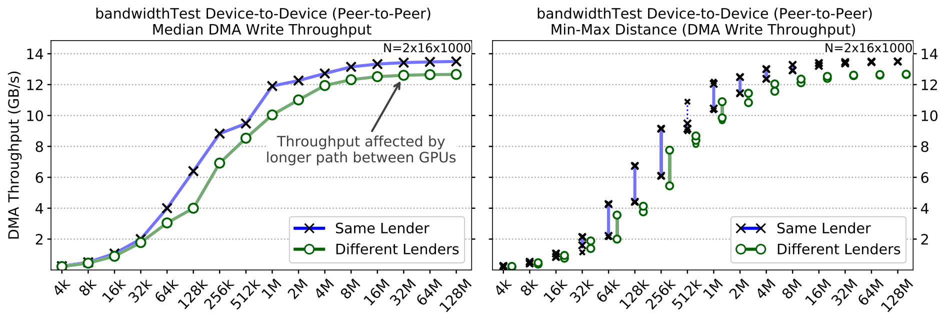
- **Same Lender**, shown in Figure 26(a): Using two GPUs from the same lender. As we established in the previous section, this scenario is similar to a local system using local devices.
- **Different Lenders**, shown in Figure 26(b): Using two GPUs from different lenders. DMA transactions have to traverse four additional hops (NTB, cluster switch, NTB, internal switch) compared to the baseline. We expect the additional latency to manifest itself as an observable performance difference when compared to the Same Lender scenario:



(a) **Same Lender**: using two GPUs from the same system. The data path of one GPU writing to the memory of the other is illustrated.

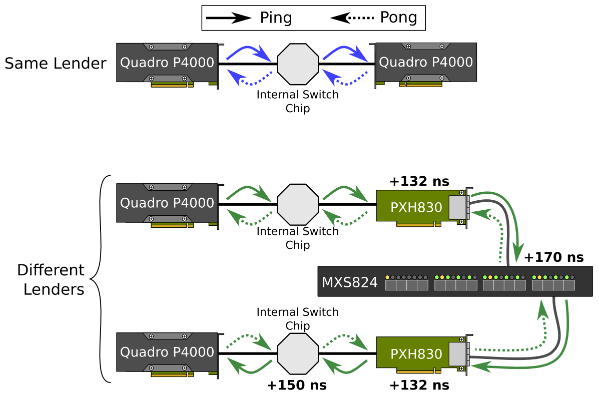


(b) **Different Lenders**: using two GPUs from different systems. The data path of one GPU writing to the memory of the other is illustrated. Note the increased number of hops.

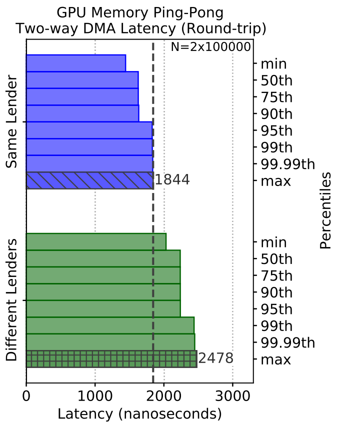


(c) The median throughout of DMA writes from one GPU to another GPU (left), and the throughput distribution of all writes shown as min–max distance (right). We can see that moving the second GPU to a different system, thus increasing the distance, affects performance.

Fig. 26. Peer-to-peer throughput: We evaluate the impact of increasing the distance between the devices.



(a) Simplified illustration of the topologies shown in Figure 26, showing the path of ping–pong DMA between the GPUs. Each hop slightly increases the completion latency.



(b) Distribution of ping-pong latency measurements. The increased distance with additional chips between the GPUs affects the latency.

Fig. 27. Peer-to-peer latency: using a ping–pong CUDA program, we measure the latency of DMA writes between two GPUs residing in different hosts. While the additional hops increase the ping–pong latency, this is expected for longer PCIe paths.

— The PEX8733 switch chip used in the PXH830 NTB adapters specifies that up to 132 ns may be added to a transaction in worst-case [13].
— The internal PEX8796 chip used internally in the expansion chassis can add up to 150 ns to transactions in worst case [14].
— Experiments in our lab show that the PM8536 PFX chip used internally in the MXS824 cluster switch adds an average latency of around 170 ns.

All hosts are running Ubuntu 18.04.2 with the 4.15 version of the Linux kernel. As before, the borrower is using CUDA 10.1 with the corresponding 418.39 version of the Nvidia CUDA driver.

Figure 26(c) shows the result of running the CUDA bandwidthTest program, copying memory from one GPU to the other using the on-board DMA engine with different transfer sizes. Figure 27(b) show the ping–pong latency using the CUDA program we described earlier. While we observe that the additional distance affects the measured throughput and back-and-forth latency, this difference is less than the worst case. This is a strong indicating that our implementation does not add any additional latency beyond what we expect from the hardware. We argue that the added latency from increasing the distance between GPUs is a reasonable trade-off with regards to increasing device utilization. It is also possible to optimize for data movement by borrowing devices that are physically close to each other in terms of number of hops, thus minimizing the distance between them. Finally, we can observe that when conditions are comparable, i.e., the PCIe path is similar, the performance is the same. We argue that this demonstrates that our Device Lending implementation does not add any overhead. After all, the speed of electrons through the silicone of the hardware is beyond the scope of our implementation.

7.1.6 Sharing SR-IOV Devices. As mentioned in Section 3.1, the term "device" actually refers to individual PCIe endpoints, or rather device functions. Some devices may implement SR-IOV, allowing a single device to virtualize multiple device functions in hardware. Each virtual function appears to the system as a separate device function with its own resources. Since our SmartIO system does not make any distinction between physical and virtual functions, it is possible to disaggregate an SR-IOV device and assign a virtual function to a remote host (without any performance penalty) the same way SmartIO distributes physical functions. Therefore, we have conducted experiments using a Mellanox ConnectX-5 100 Gigabit Ethernet adapter, which supports up to 1024 virtual functions [81]. Each virtual function implements a (virtual) Ethernet controller. By generating high network throughput and comparing the performance of a virtual function to the performance of the physical function, for both a local system and a remote system using Device Lending, we argue that this will reveal any hidden performance overheads caused by our implementation that could affect hardware virtualization.

To create network workload and generate network traffic, we have used the *iperf2* tool. This tool is widely used for measuring network performance, and is available on most Linux distributions. iperf2 supports creating TCP data streams between a *client*, running on a local host, and a *server*, running on a remote host. The client writes as much data to the TCP stream as it is able to, and the server reads from the stream.[11] In this respect, TCP is designed to provide a reliable data stream over a lossy IP network where the kernel is involved in encapsulating raw data into TCP segments and IP packets, managing transmission and receive buffers, handling retransmissions and flow control, and network congestion avoidance—all of which require CPU time. Therefore, to fully saturate a 100 Gigabit link without becoming CPU-bound, iperf2 supports spawning dedicated threads for each individual TCP connection on both the server and the client. Each individual thread can run on its own CPU core.

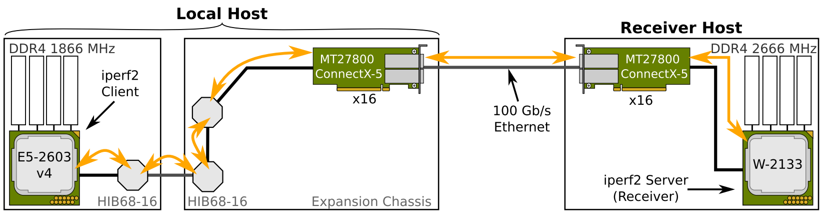
Figure 28 depicts the configuration used in these tests, where the client connects to the server running on the receiver host:

- **Local Baseline**, shown in Figure 28(a): A local system using its local network adapter to connect to the dedicated Receiver Host, running the iperf2 server. The iperf2 client is running on the local CPU. We ran one test using the adapter's physical function and one test using one of the adapter's virtual functions, to rule out any performance overhead caused by the virtualization.
- **Device Lending**, shown in Figure 28(b): A borrower using a remote network adapter to connect to the dedicated Receiver Host. As with our Local Baseline tests, we borrowed first the physical function and then the virtual function, to rule out any performance difference.

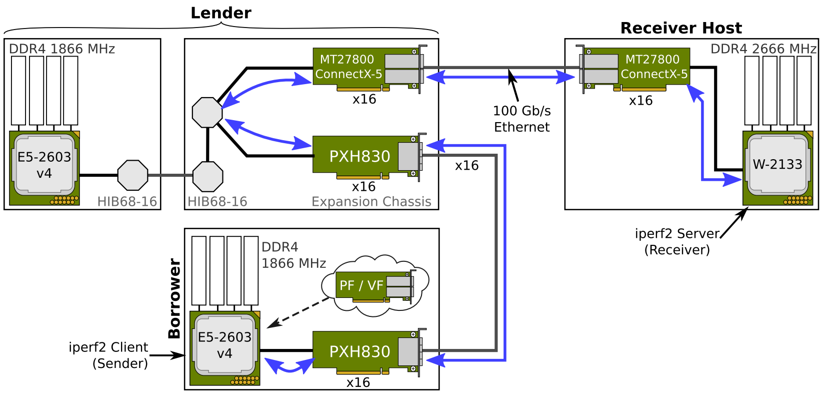
All hosts run Ubuntu 18.04 with the 4.15 version of the Linux kernel, using the in-kernel Mellanox Ethernet driver. To compare apples to apples, we have disabled the IOMMU on both lender and borrower, as well as on the receiver host. In all cases, the iperf2 client runs for a duration of five minutes, writing to the TCP streams and reports the throughput every half-second. The client and the server were configured to use four parallel connections, and, consequently, using four threads each. We relied on the default kernel scheduler to schedule threads on different CPU cores. We also experimented with various network related settings in the kernel, such as increasing buffer sizes and using alternative TCP congestion control mechanisms. Additionally, we tried different offloading mechanisms supported by the adapter. However, besides setting the Ethernet maximum transfer unit to 9,000 bytes ("jumbo frames"), the default 4.15 kernel settings and disabling all forms of offloading provided highest throughput.

Figure 28(c) shows the throughput measurements of our comparison, with performance for a physical function shown on the left (PF), and performance for a virtual function on the right (VF). Note that while it is common to describe network performance in terms of Giga*bits*, we have

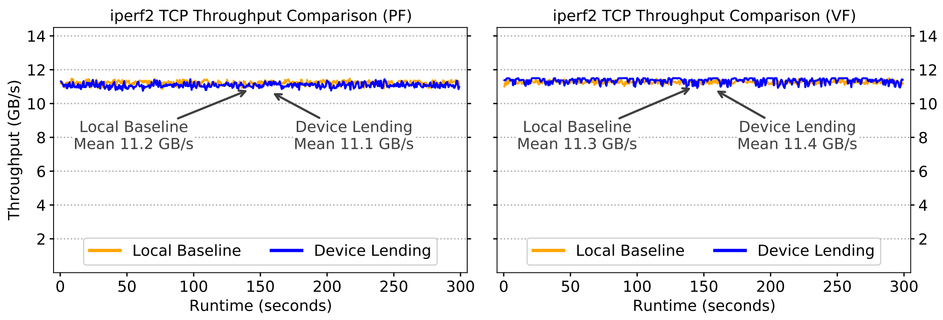
[11]The behavior of iperf2 is perhaps counter-intuitive. In most client/server applications, the client will typically request data from the server rather than the server acting as a receiver. We have used the same terminology as the program uses.



(a) **Local Baseline**: using a local network adapter to write data to a receiver over TCP.



(b) **Device Lending**: using a borrowed (remote) network adapter to write data to a receiver over TCP.

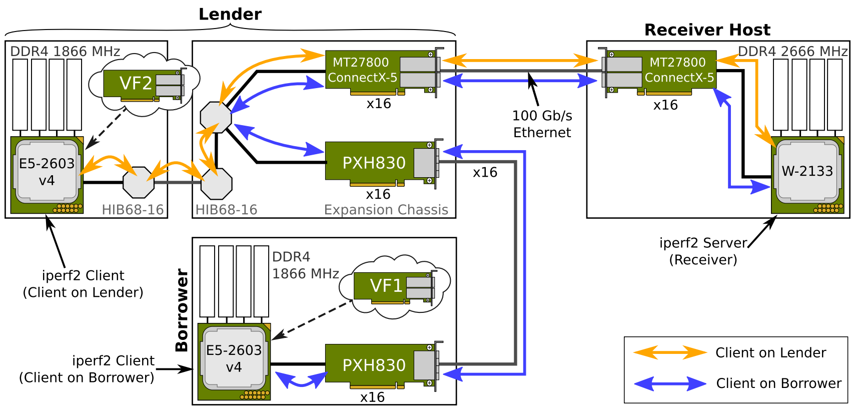


(c) TCP throughput at 1/2 second intervals for both the physical and virtual functions. Observe that there is no significant performance difference between the Local Baseline and Device Lending. Additionally, hardware virtualization (VF) does not impact performance compared to the physical device function (PF).

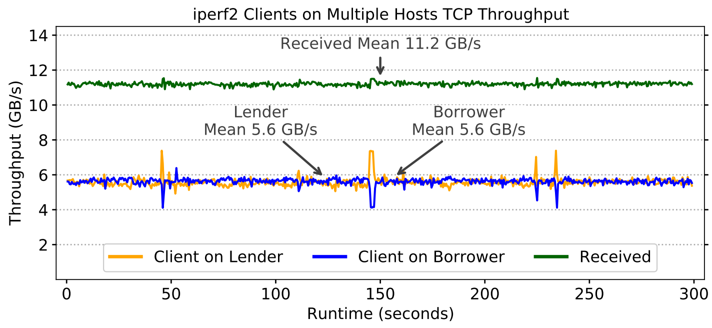
Fig. 28. TCP throughput comparison: We compare the achieved throughput for a client/server application.

plotted performance in terms Giga*bytes* to be consistent throughout this article. By comparing the performance of these functions being used locally (Local Baseline) and remote (Device Lending), we prove that accessing a borrowed virtual function does not introduce any performance overhead. Additionally, we also observe that for the Mellanox adapter used in this experiment, there is no measurable difference when using a virtual function compared to using a physical function.

Moreover, multiple hosts can share the same device by distributing individual virtual functions. Since most SR-IOV-capable devices support several virtual functions, this becomes highly useful



(a) By configuring two virtual functions and assigning one to the local host and one to the remote borrower, the Lender and the Borrower can connect to the Receiver Host simultaneously through the same adapter.



(b) TCP throughput at 1/2 second intervals for both iperf2 clients. We also show the received data rate on the server, which is the combined data rate from both clients. While throughput fluctuates somewhat, both clients' transmission rate equilizes over time.

Fig. 29. Two hosts using the same SR-IOV-capable network adapter simultaneously.

with regard to our SmartIO system. To demonstrate this, we have performed an additional test where the lender and the borrower share the same sender-side network adapter simultaneously, to transmit data to the receiving server. Figure 29(a) shows the topology of this multi-host sharing test. We configured two virtual functions for the network adapter and assigned them to the two hosts: One function is used locally by the lender, and we run an iperf2 client on the lender with two parallel connections (threads) to the iperf2 server (Client on Lender). The other function is used simultaneously by the borrower, and we run an iperf2 client on the borrower as well, also using two threads (Client on Borrower).

Figure 29(b) shows the results of our multi-host test, where we have plotted the reported throughput for both clients. The server's reported received data rate, which is the combined rate of the two clients, is also shown. While throughput for the two clients fluctuate a little, they approach the same throughput over time (as can also be seen by comparing the mean throughput). This is expected behavior for TCP streams, as they alternate between increasing transmission rate in an attempt to estimate the available network bandwidth, and backing off when they exceed their fair share of the total capacity.

Finally, it should also be mentioned that sharing the Mellanox network adapter does not only provide connectivity to the receiver host for both lender and borrower, but it also becomes possible for the lender and borrower to establish IP connections to *each other* as well. In a larger PCIe cluster, this could be useful for IP network applications that could communicate with each other, using only a single network adapter and without sending a single packet out on the Ethernet link.

7.2 Scaling Heavy Workloads

Another method of demonstrating that there is no hidden overhead in our Device Lending implementation, is investigating how it behaves under stress. It might be the case that there are small overheads caused by the implementation that only become visible when the system is under heavy load. Because of this, we have also designed an experiment using a realistic GPU-intensive machine learning workload, to prove that Device Lending is a solution for composable and disaggregated PCIe infrastructure suitable for real-world applications.

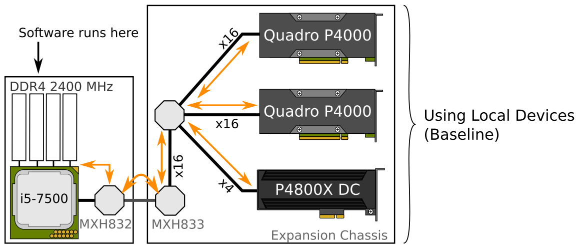
Our workload is a typical convolutional neural network training using the Python machine learning framework Keras [1]. Keras is a high level framework that wraps different lower level machine learning frameworks. In our case, Keras uses Tensorflow [2] as its back-end. Keras also allows multiple GPUs to work together, by replicating the machine learning model being trained on each of the GPUs, and splitting the model's inputs into "sub-batches" and distributing them on the GPUs. When the GPUs are done, the sub-batches are concatenated on the CPU into one batch. This introduces quasi-linear speed-up. We used Python 3.6 and Keras 2.2.4, running on Ubuntu 16.04 (4.9 kernel) with CUDA 9.0 and cuDNN 7.1 in our tests.

We wrote a program that trains available models in Keras on given datasets with given hyperparameters using transfer learning [57]. In our case, we use a VGG19 [76] model that is pre-trained on the ImageNet dataset [20], and the model was re-trained using an 8-classes image dataset of the gastrointestinal tract called Kvasir [30, 63, 64] to perform disease classification [65].

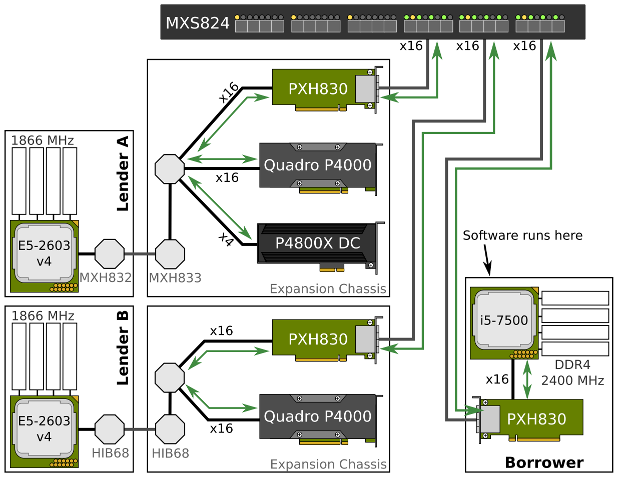
We measure the runtime of 12 epochs of the model training on two Nvidia P4000 GPUs as well as loading images from storage and writing the results back using an Intel Optane P4800X NVMe device. While 12 epochs may not give the statistical significance needed for reliable machine learning results, we are only interested in system performance. Both GPUs and the NVMe were used in all scenarios. Figure 30 shows the scenarios and results of our experiment:

- **Local Devices**, shown in Figure 30(a): A local system using both GPUs and the NVMe device locally. This scenario serves as our baseline comparison. The IOMMU is disabled, to allow peer-to-peer transactions between the GPUs.
- **Single Lender** (not depicted): A borrowing system connected back-to-back and accessing all three devices remotely. The number of hops in the path is similar to the Local Devices scenario. The IOMMU on the lender is disabled, while it is enabled on the borrower to shrink the DMA window size down to 1 GB. We can see from the results in Figure 30(c) that this scenario achieves approximately the same epoch runtimes as the local comparison scenario, demonstrating that there is no hidden overhead in our Device Lending implementation.
- **Two Lenders**, shown in Figure 30(b): A borrowing system accessing devices from two separate lenders. The IOMMU on the borrower is enabled, while it is disabled on both lenders. As the GPUs reside in different hosts, the path between them increases. This appears to slightly affect the epoch runtimes, as seen in Figure 30(c).

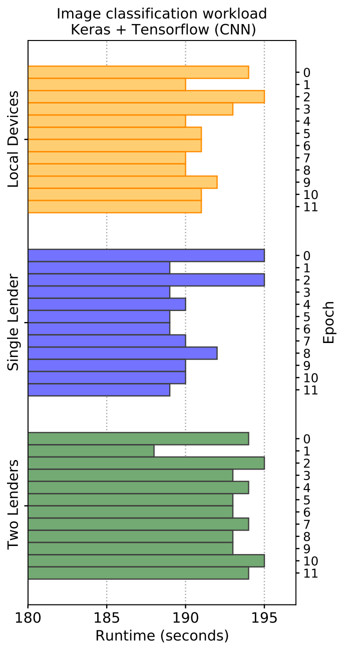
Our machine learning workload proves it is possible to use Device Lending for realistic workloads in a PCIe cluster, dynamically creating configurations of both local and remote devices and accessing them without any performance penalty beyond what is expected for longer PCIe paths. We argue that this effectively demonstrates the capacity of our implementation for creating



(a) **Local Devices**: A local host using two local GPUs and a local NVMe device.



(b) **Two Lenders**: A host borrowing a remote GPU and an NVMe device from one lender and a borrowed GPU from a different lender.



(c) Runtime of each individual epoch for all scenarios.

Fig. 30. Scaling heavy workloads: We demonstrate the usability of SmartIO for composable and disaggregated PCIe infrastructure, by comparing the performance of running a GPU-intensive machine learning workload on a local system using local devices to Device Lending using remote devices. As data is moved between the GPUs, the increased distance between them affects the total runtime. However, we can see that when the devices reside in the same host, our Device Lending implementation does not add any measurable overhead.

a disaggregated PCIe infrastructure that supports dynamic scaling of devices that are distributed in the cluster.

7.3 VM Pass-through with MDEV

While VFIO pass-through enables direct access to *local* physical devices from a VM guest, our MDEV pass-through mechanism enables direct access to *remote* devices. However, our MDEV extension to KVM requires the use of an IOMMU on the lender to map the device into the same guest-physical address space as the VM as explained in Section 5.2. This effectively disables shortest-path routing in the fabric, as transactions must be forwarded through the CPU on the lender in order for the IOMMU to resolve virtual addresses to physical addresses. Intuitively, we expect this to cause some performance degradation.

7.3.1 IOMMU Performance Penalty. Processor designs are complex and often not well-documented, making it difficult to determine what exactly happens with memory transactions in progress once they leave the PCIe root complex and enter the CPU. Memory transactions may be buffered while awaiting IOMMU translations, or the IOMMU may need to perform a multi-level table look up for resolving addresses.

To distinguish between overhead caused by our software implementation and any overhead caused by the hardware address virtualization, we compare the performance of the MDEV implementation to bare-metal performance using Device Lending. As described in Section 4.3, Device Lending includes optional IOMMU support allowing us to isolate the performance penalty of enabling the IOMMU. As such, this establishes a baseline we can compare our MDEV implementation with. Note that our exhaustive evaluations of Device Lending presented in Section 7.1 demonstrate that the Device Lending mechanism does not add *any* performance overhead compared to native access. Therefore, we argue that Device Lending a valid bare-metal comparison to our MDEV implementation to reveal any overhead caused by MDEV.

Two hosts are connected back-to-back with Dolphin PXH830 NTB adapters, and we use the same One Stop Systems expansion chassis as our previous tests. We installed an Nvidia Tesla K40c GPU alongside the NTB adapter in the chassis. The expansion chassis is connected upstream using Dolphin MXH832 host and MXH833 target transparent adapters. By turning the IOMMU on the lender on and off, we are able to compare the performance difference of address virtualization on peer-to-peer DMA transfers over the NTB. By using the expansion chassis, we are able to create a worst-case scenario for enabling the IOMMU, as the distance between the devices and the CPU increases. Figure 31(a) depicts the three scenarios compared in this evaluation:

- **Bare-metal No-IOMMU**, where we use Device Lending to facilitate direct access to the remote GPU. The IOMMU on the lender is turned off to enable shortest-path routing within the expansion chassis. Since the GPU is unable to reach high I/O addresses, we enabled the borrower-side IOMMU and configured the DMA window size to 512 MB. We also made sure that the bandwidthTest program ran with the same CPU core affinity as the local NTB adapter.
- **Bare-metal IOMMU** is similar to the No-IOMMU scenario in every way, except that lender-side IOMMU is enabled. By using the lender's IOMMU, we are able to configure larger DMA windows while still setting up mappings over the NTB for the GPU using low addresses. Note that since we are using the expansion chassis, this becomes the aforementioned worst-case scenario for Device Lending; all transactions must be routed towards the lender's CPU so that the IOMMU can resolve virtual I/O addresses. As with the No-IOMMU scenario, we made sure to run the bandwidthTest program with the same CPU core affinity as the local adapter.
- **MDEV:** We also installed Qemu 2.10.1 on the local host and configured it to use the KVM hypervisor. Using our MDEV extension to KVM, we borrow and "pass through" the GPU to the VM guest, enabling direct hardware access to the guest driver. The VM was configured to have 4 GB memory, and we used 2 MB "huge pages" on the host. Our MDEV implementation probes the VM for low and high guest physical memory dynamically, and sets up respective DMA windows. Because of this, we need to configure the NTB BAR size to be larger than the VM memory. Finally, we also made sure that Qemu ran with the same CPU core affinity as the local NTB adapter.

We installed Ubuntu 16.04 with the 4.10 version of the Linux kernel on both machines, as well as the guest OS in the VM. Although Device Lending is currently only supported on Linux, any guest OS would have been possible, including Microsoft Windows. However, we chose to use same version of Linux as both host and guest OS, to run as similar software as possible in all scenarios. CUDA version 9 was installed on the local host and in the VM guest. We used the bandwidthTest program described in Section 7.1, to measure the throughput of DMA writes and DMA reads to system memory using the GPU's own on-board DMA engine. As with our previous evaluations, bandwidthTest was configured to do 1,000 iterations for each transfer size from 4 kB to 128 MB.

Figure 31(b) shows the median DMA read and write throughput for all three scenarios. We observe that the throughput drops significantly when the IOMMU is enabled, particularly for reads (drops from 10.2 GB/s to just a little over 1.5 GB/s. There are two primary reasons for this significant performance drop:

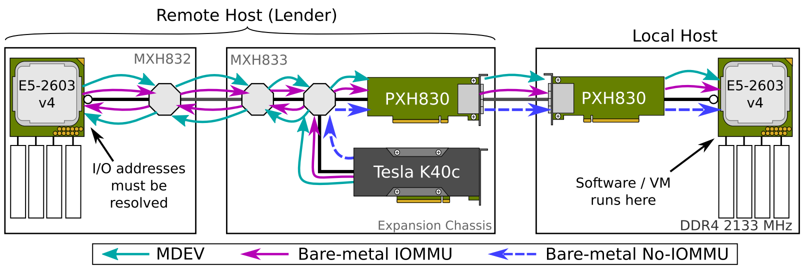
(1) Reads suffer particularly from the increased distance, as addresses are routed through the lender's CPU *twice* per transaction; the first time in order for the IOMMU to translate the addresses of the read requests, and the second time for completions with the requested data.
(2) By using a PCIe tracer, similar in concept to that of network packet tracers, we were able to investigate what the actual transactions look like on the fabric. By first using the tracer in the GPU slot, and then in the lender-side NTB slot, we were able to observe that the transactions are modified by the Intel Xeon CPU used in our test; the GPU requests 256 bytes per request, but each request is emitted as 4×64 byte requests on the other side of the IOMMU. As the CPU is only able to keep a limited number of non-posted requests open at the same time, splitting up read requests into multiple smaller requests leads to very poor link utilization.

Regardless, by comparing the bare-metal scenario with the IOMMU enabled to MDEV, we observe that the performance of DMA transfers is almost identical for both scenarios. While the performance drops because of the increased paths and IOMMU address translation, our results indicate that our MDEV implementation does not add any overhead on top of the hardware virtualization.

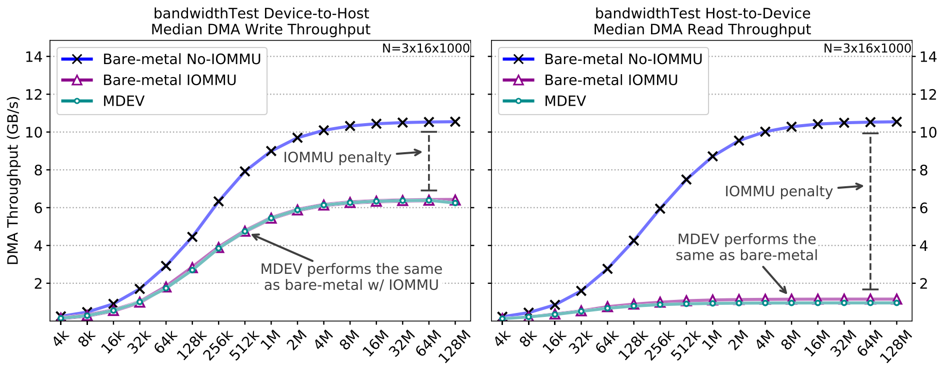
7.3.2 Pass-through Comparison. We have also repeated the same peer-to-peer benchmarks described in Section 7.1 using VMs. By using the peer-to-peer benchmarks to measure throughput and latency between two GPUs, we are able to compare our MDEV extension using remote devices to "normal" VFIO pass-through on a local system.

Figure 32 shows the topologies used in our comparison evaluation:

- **Local VFIO**, shown in Figure 32(a): A Qemu 2.10.1 instance running on a local system using the KVM hypervisor. By using VFIO, we pass-through two local Nvidia Tesla K40c GPUs. The local IOMMU is enabled, in order for KVM to map the devices into the same guest-physical address space as the VM. The guest OS is Ubuntu 6.04 with the 4.10 version of the Linux kernel, and we are using CUDA version 9. The host OS is Fedora 29 using the 4.18 version of the kernel.
- **MDEV**, shown in Figure 32(b): A Qemu 2.10.1 instance using the KVM hypervisor and our MDEV extension to borrow and pass-through two remote GPUs from the lender. We used the same OS image for the VM as the VFIO scenario, and Fedora 29 on the hosts. The lender's IOMMU is enabled, as is required by MDEV.
- **Bare-metal**, shown in Figure 32(b): We also include a bare-metal baseline, running bandwidthTest natively on a bare-metal machine using Device Lending. Two remote GPUs are borrowed by a bare-metal machine. The bare-metal borrower machine boots the same OS image as we used for our VMs. On the lender, we ran Fedora 29. The lender's IOMMU is enabled, to make the data path comparable to MDEV.



(a) Transactions must be routed through the CPU in order for the IOMMU to resolve addresses.



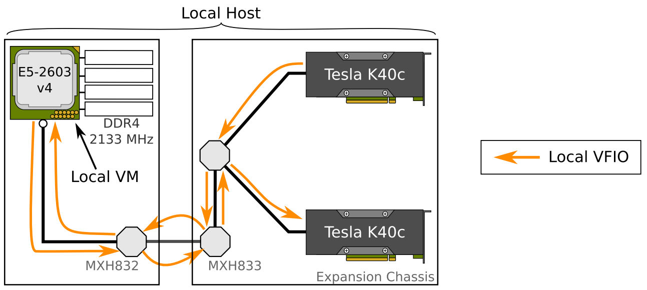
(b) Median throughput of DMA writes (left) and DMA reads (right). While enabling the lender's IOMMU significantly decreases performance, note that our MDEV implementation achieves the same performance as the bare-metal comparison (with the lender-side IOMMU enabled). This indicates that our implementation does not add any overhead in addition to hardware virtualization.

Fig. 31. IOMMU performance penalty: By using the IOMMU on the lender, shortest path routing is disrupted.

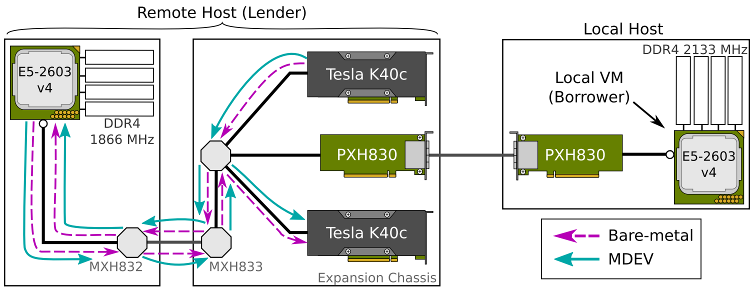
Both VM instances were configured with 4 GB memory, and we enabled 2 MB huge pages on the host. We also set the CPU affinity to be the same as the local adapter in both the bare-metal and MDEV scenarios.

Like before, we configured bandwidthTest to copy memory from one GPU to another using transfer sizes from 4 kB to 128 MB. Figure 33(a) shows the median throughput (left) and throughput distribution as a min–max distance (right). Each transfer size is repeated 1,000 times, and we have marked measurements below the 0.2th percentile as outliers. We observe that the local VFIO pass-through scenario reports a slightly higher throughput than both our MDEV implementation and the bare-metal comparison(!) for smaller transfer sizes.

In order for the GPU to notify the host driver that the DMA transfer is complete, it relies on interrupts. The bandwidthTest program measures throughput by initiating a memory copy (DMA transfer) and recording the time elapsed until the transfer is complete. As KVM uses a different mechanism for notifying the VM guest about an interrupt for VFIO pass-through devices than our MDEV implementation, we speculate that interrupts raised by VFIO pass-through devices may cause KVM to briefly suspend the execution of Qemu to handle the interrupt and signal *eventfd* events. This would in turn would affect timing measurements by software running in the VM. However, as the measured throughput converge for all three scenarios when the transfer size increases, this suspected measurement discrepancy seems to become less significant.



(a) **Local VFIO**: two GPUs being used by a local VM with KVM+VFIO pass-through. The direction of DMA writes from one GPU to the other is shown.



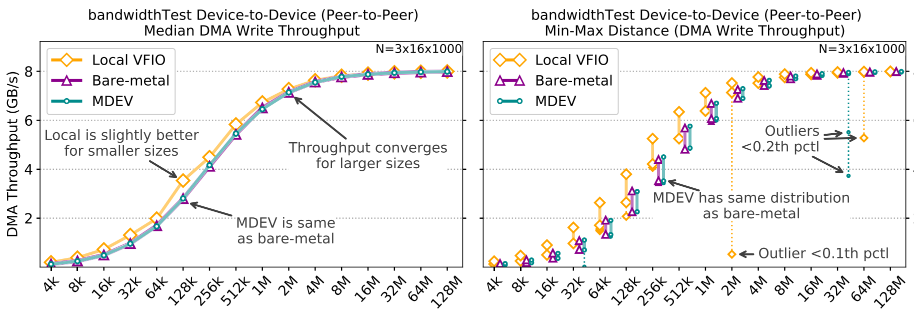
(b) **Bare-metal** and **MDEV**: two GPUs being used by a remote bare-metal machine with our Device Lending implementation and by a remote VM with our KVM+MDEV pass-through implementation. The direction of DMA writes from one GPU to the other is shown. Note that the data path is similar to the local VFIO scenario.

Fig. 32. Peer-to-peer topologies: We compare the measured throughput and latency between two GPUs passed through to a VM using local VFIO pass-through to using our MDEV pass-through of remote devices. Note that we have also included a configuration using bare-metal Device Lending.

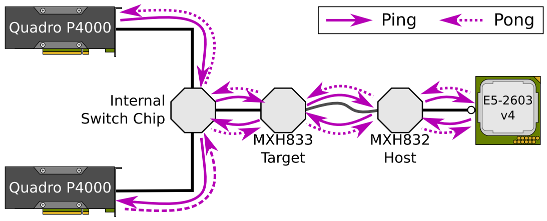
Figure 33(c) shows the distribution ping–pong latency measurements using the CUDA program we described in Section 7.1.4, where two GPUs writes a counter back and forth to each other's memory. The maximum measurement for MDEV appears to be an outlier, so we have annotated the 99.99th percentiles instead. The distributions for MDEV and bare-metal are similar, indicating that our MDEV implementation does not add any additional overhead beyond hardware virtualization. Unlike the bandwidthTest program, which uses device interrupts for synchronizing timing measurements, the ping–pong measurements use elapsed clock cycle for recording time (as described in Section 7.1.4). With this method, it appears that the strange effect where VFIO performs better than bare-metal is not present, which strengthens our suspicion that it is related to delivering interrupts to the VM.

7.4 Distributed NVMe Driver Evaluation

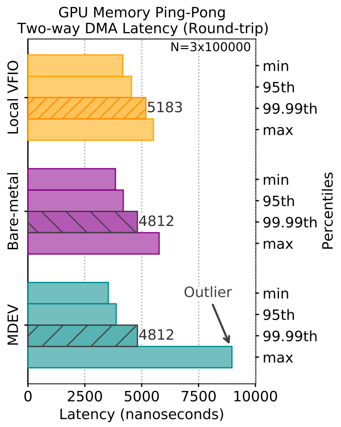
Our Device Lending and MDEV extension make it possible for a local device driver to operate a remote device in a manner that is fully transparent to both device and driver. This is possible as we prepare memory mappings in advance and inject addresses that map over the respective NTBs. However, as the physical memory allocated by a device driver or a VM instance is outside of our control, we are forced to map *all* of local memory for a remote device. As we have seen in Sections 7.1.4 and 7.1.5, increasing the distance PCIe transactions has to travel has



(a) Throughput distribution of DMA writes from one GPU to the other. We observe that while our MDEV implementation does not add any overhead compared to the bare-metal comparison. Local VFIO pass-through performs slightly better for small transfer sizes, but the throughput converges for larger transfers. Note that the Y-axis is adjusted for the lower throughput of peer-to-peer DMA using the IOMMU.



(b) Simplified illustration of the topologies shown in Figure 32, showing the path of ping–pong DMA between the GPUs. Note that the path is the same for all three scenarios.



(c) Distribution of ping–pong latency measurements. Note that MDEV performs the same as bare-metal.

Fig. 33. Peer-to-peer evaluation: Using the same bandwidthTest and ping–pong CUDA programs as previous evaluations, we measure both throughput and latency of DMA writes between two GPUs. Our MDEV implementation does not add any overhead compared to bare-metal.

an impact on performance. Particularly non-posted transactions, such as reads, are affected by longer distance between requester and completer. In other words, increasing the distance between the borrower and the device will negatively impact performance, as the distance between the device and the memory it accesses also increases.

However, a programmer can fully exploit shared memory capabilities in PCIe clusters by using the SISCI API [22]. Local memory may be exported for other nodes, and remote memory can be mapped for the local application. It is even possible for a node to allocate memory buffers on local devices, such as GPUDirect-capable GPUs, and other nodes to map this memory through their own NTBs.

Our SmartIO device driver extension to SISCI aims to combine the best of both worlds. Device drivers can remain agnostic about the local address space in the node where the device physically resides as our SmartIO system resolves local I/O addresses. Simultaneously, drivers may fully exploiting shared memory capabilities of the PCIe network by building on top of the existing SISCI functionality. The trade-off is that existing device drivers must be modified or rewritten to use this new API extension. In an attempt to make the case for why this trade-off might be worthwhile, we have evaluated the latency benefit our proof-of-concept distributed NVMe driver.

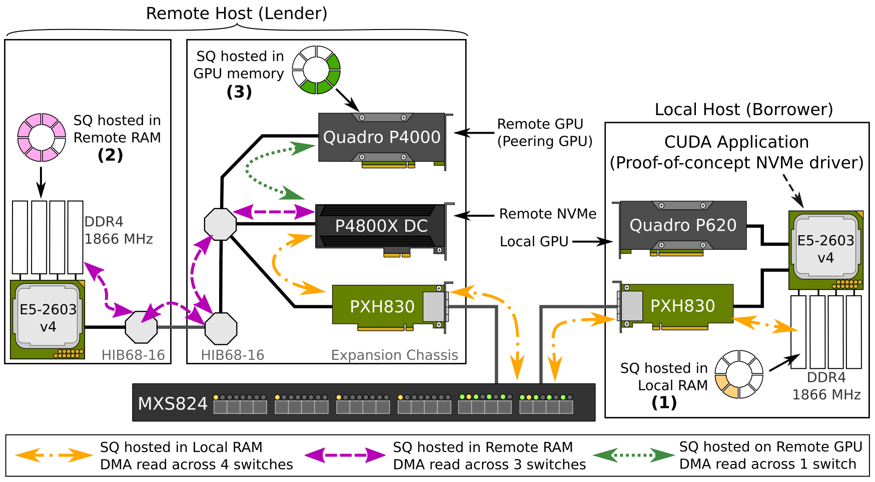
7.4.1 Optimizing Data Access Patterns. We outlined our userspace NVMe driver implementation using the SmartIO SISCI API extension in Section 6. Not only are we able to assign individual queues to different nodes, but we are also able use GPUDirect-capable GPUs to host queues in GPU memory as explained in Section 6.4. Since it is possible to combine the SmartIO API extension with borrowed GPUs (using Device Lending), we can design truly elastic workloads. *Any* type of (linear) memory, such as RAM or device memory, may be exported and made available for a cluster application, whether it runs on a CPU, a GPU, or another PCIe computing accelerator—or even a combination of CPUs, GPUs, and accelerators.

To avoid reading over long distances in the cluster, we can use this flexibility to facilitate moving data around in the cluster by using a "push" strategy instead. The NVMe standard does not have any restrictions regarding memory locations for paired queues; from the NVMe device's point of view, any address it can use DMA to is potentially a valid queue memory location. This means that we can allocate an SQ in memory close to the device, while allocating the associated CQ in memory close to the CPU that polls it. As explained in Section 6.1, our API extension supports specifying access pattern hints when allocating memory segments. By specifying that the CQ segment will be mostly read from by the CPU and only written to by the device, the CQ memory segment will be allocated in the borrower's local memory. Similarly, by specifying read access by the device (and only write access by the CPU) for the SQ memory segment, our SmartIO driver API will prefer memory close to the NVMe. As PCIe provides us with an ordering guarantee, the CPU or GPU may simply write the command to remote memory and immediately after ring the doorbell register.[12] This means that when the NVMe device is notified by the doorbell write, we can be certain that the command has arrived in the queue, and the NVMe may read it using DMA.

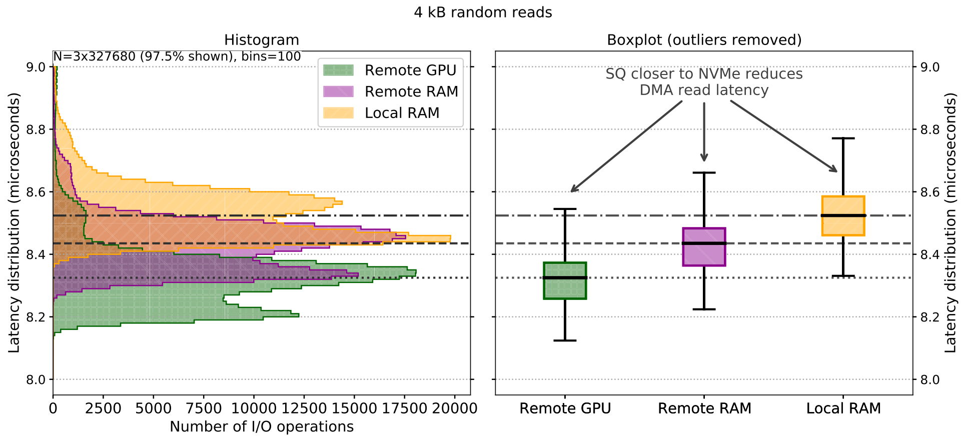
To evaluate the performance benefit of this strategy, we have designed the following experiment: a local CPU runs our proof-of-concept userspace NVMe driver (implemented as a CUDA application). It uses a local Nvidia Quadro P620 GPU and a remote Intel Optane P4800X DC NVMe device. The local GPU is managed by the native CUDA driver, while the remote NVMe device is operated by our application (proof-of-concept driver). The application reads data from the NVMe directly into GPU memory on the local GPU. Note that "local" and "remote" in this experiment refer to the CPU the application runs on. The NVMe CQ is allocated in the borrower's local RAM, while we have used three different memory locations for placing the SQ as shown in Figure 34(a):

(1) **SQ hosted in Local RAM:** We allocated queue memory for the first SQ in local RAM, and mapped this for the NVMe device. When the application rings the doorbell register, the NVMe has to read across 4 hops along the path, including internal PEX8796 switch chip in the expansion chassis, the PM8536 PFX switch chip used internally in the MSX824 cluster switch, as well as the PEX8733 chips used in the PXH830 NTB adapter cards.
(2) **SQ hosted in Remote RAM:** The memory for the second SQ was allocated in remote memory, i.e., RAM on the lender. As we use the same expansion chassis as previous evaluations with HIB68-16 transparent adapters, the NVMe has to read across 3 hops when the application rings the doorbell, including the internal switch chip in the expansion chassis and the PEX8733 chips used in the HIB68-16 transparent adapter cards.

[12]NVMe I/O commands are 64 bytes, so writing a command will automatically flush the Write-Combining Buffer on x86.


Remote Host (Lender)
SQ hosted in GPU memory (3)
SQ hosted in Remote RAM (2)
DDR4 1866 MHz
E5-2603 v4
HIB68-16
HIB68-16
Quadro P4000
P4800X DC
PXH830
Expansion Chassis
Remote GPU (Peering GPU)
Remote NVMe
Local GPU
Local Host (Borrower)
CUDA Application (Proof-of-concept NVMe driver)
Quadro P620
E5-2603 v4
PXH830
SQ hosted in Local RAM (1)
DDR4 1866 MHz
MXS824
SQ hosted in Local RAM DMA read across 4 switches
SQ hosted in Remote RAM DMA read across 3 switches
SQ hosted on Remote GPU DMA read across 1 switch

(a) Our SmartIO NVMe driver makes it possible to allocate memory in different locations in the cluster, even on GPUDirect-capable GPUs, and use this memory for I/O queues. We have measured the I/O command completion latency for three scenarios: (1) hosting the SQ in RAM on the borrower, (2) hosting the SQ in RAM on the lender, and (3) using memory of a peering GPU to host the SQ.



(b) I/O command completion latency distributions as a histogram (left) and as a boxplot (right). The median for all three distributions are marked with horizontal lines. We observe that the closer the SQ is to the NVMe device, the lower the completion latency is.

Fig. 34. SQ placement: We evaluate the impact of moving the SQ closer to the NVMe device. By reducing the distance the NVMe device has to read to fetch I/O commands, we are able to reduce the command completion latency.

(3) **SQ hosted in Remote GPU memory:** Using Device Lending, we also borrowed an Nvidia Quadro P4000 GPU from the same lender and allocated memory for the third SQ as a memory buffer on this GPU. While the borrowed GPU is operated by the local CUDA driver, both Device Lending and the SmartIO API extension uses the same underlying SmartIO system, so mapping this memory for the NVMe device uses the same address resolving mechanism described in Section 4.4. As the GPU is installed next to the NVMe device in the same

expansion chassis, the NVMe only has to read through the internal switch chip in the expansion chassis. Note that GPU memory has different memory characteristics than system RAM.

Both hosts are running Ubuntu 18.04.4 with the 4.15 version of the Linux kernel, and the local host (borrower) is running CUDA 10.2 with the included GPU driver. The IOMMU is enabled on the local host, while it is disabled on the remote host (lender) to use shortest-path routing. While not strictly necessary for this experiment, we also enabled persistent mode on both GPUs.

For each SQ location, one by one, our application executes 327,680 NVMe read commands of 4 kB chunks of data from storage each, starting at a pseudo-random offset for each chunk. The *command completion latency* for each single command was recorded, and we used a queue depth of just one entry to avoid aggregated measurements. We define the command completion latency as the time elapsed between the driver writing a command to the SQ, followed by a write to the doorbell register, until the corresponding completion shows up in CQ memory (local memory). As we start the timer before writing the command, part of the latency measurement is the time it takes to write to (remote) memory. Note that our NVMe driver implementation uses polling instead of relying on interrupts, and that the data is written by the NVMe directly into memory onboard the local GPU using peer-to-peer DMA.

Figure 34(b) depicts the distributions of latency measurements for all three SQ placements. The same datasets are shown as both a histogram (left) and as a boxplot (right). Note that we have adjusted the Y-axis, so outliers are not shown. Our results demonstrate that moving the SQ memory closer to the NVMe device significantly reduces latency, as the distance that the NVMe device has to read across shrinks. We argue that this indicates that while there is a development cost of implementing device drivers using the SmartIO API extension, the reward is improved performance over Device Lending and native device drivers. There is also the added benefit of being able to fully utilize PCIe clustering capabilities to implement functionality such as streaming data directly into GPU memory.

Finally, it should be noted that the NVMe standard specifies optional support for one or more **controller memory buffers (CMBs)** [55]. CMBs are BARs with generic device memory that an NVMe driver may read from and write to. The intention of CMBs is that becomes possible for a driver to host queue memory on the NVMe device itself, elminiating the need for the NVMe controller to use DMA to fetch commands entirely. While the Intel Optane P4800X DC NVMe device used in our experiments does not support CMB, implementing support for it to move queues *as close as possible* to the NVMe would be trivial. Our SmartIO system automatically export device BARs as mappable memory segments, so supporting CMB would be a matter of mapping the BAR and setting up the necessary descriptors in CMB memory.

7.4.2 Sharing a Single-function NVMe Device. Due to the complexity of implementing SR-IOV in hardware, NVMe devices with SR-IOV support are not widely available. Most NVMe devices on the market are single-function devices. However, the inherent parallel design of the NVMe standard provides us with great flexibility. Each queue has its own dedicated doorbell register, which avoids contention. Pairs of SQs and CQs can operate completely in parallel, making it possible to distribute queue pairs to different nodes in the cluster using the SmartIO API extension, as explained in Section 6.2. As such, we can treat a non-SR-IOV device as a shared resource by using our NVMe driver implementation.

To demonstrate this, we designed an experiment in a larger cluster of nodes. The MSX824 cluster switch has 24 × 4 Gen3 ports that can be configured to ×8 and ×16 links by grouping two or four ports, respectively. This makes it possible to create a cluster of 60 nodes by connecting seven MSX824 switches in cascade (one top switch with six subswitches). Each individual node is connected to one of the subswitches through a x8 Gen3 link. One node was dedicated as lender,

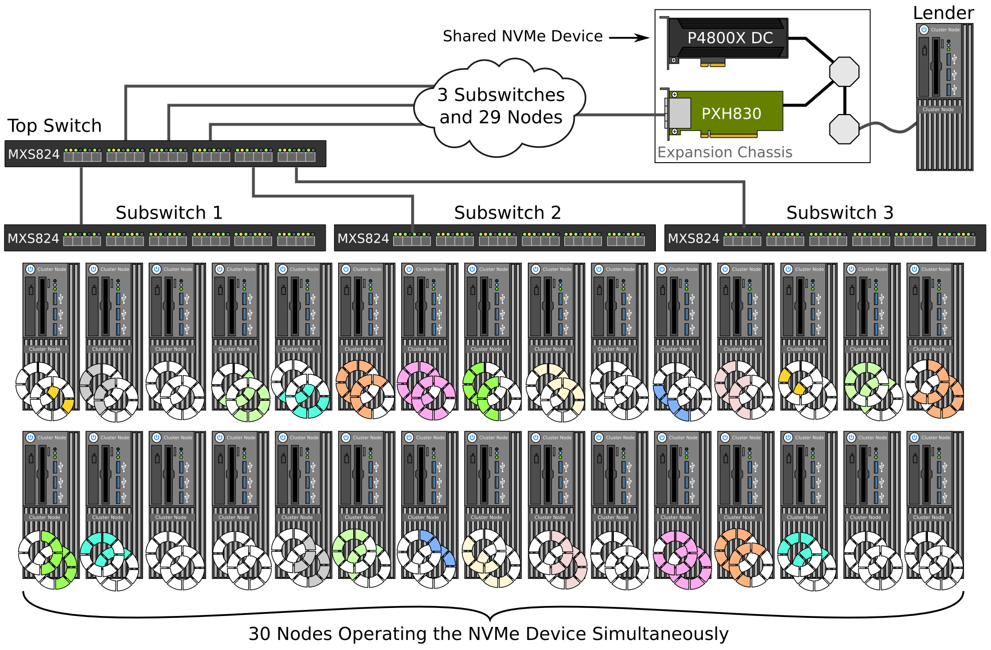


Fig. 35. By distributing an SQ and a CQ to 30 cluster nodes, we demonstrate that it is possible to concurrently share a single-function NVMe device in a larger cluster.

and was configured with an expansion chassis with the NTB adapter and an Intel Optane P4800X DC NVMe device as illustrated in Figure 35. Using our 60 node cluster setup, we performed two experiments:

(1) **Simulatenous sharing:** The P4800X NVMe used in our experiment supports up to 32 queue pairs (one queue pair is reserved for admin queues). We configured the lender to be the NVMe manager, setting up the admin queues and resetting the device, and we configured 30 other nodes to act as NVMe clients as described in Section 6.2. Each of the 30 clients configured one SQ and one CQ, allowing them to operate the NVMe independently of the other nodes, as illustrated in Figure 35. All 30 nodes each read chunks of 4 kB data in a loop, demonstrating that our queue-distribution mechanism works.

(2) **Multicast:** We configured all 59 nodes (all nodes excluding the lender) to subscribe to the same multicast group, allocating a buffer in their local memory and setting up multicast mappings. We then used one of the nodes to initiate an NVMe identify command using the address of the multicast segment. This replicated 4 kB of controller information to the memory of all 59 nodes in a *single* operation.

While number of switches in the path increases command completion latency (as is expected), hosting queues in the lender's RAM rather than in memory on the borrowers would provide a latency benefit similar to what we observed in Section 7.4.1. However, since the number of simultaneous borrowers is limited by the number of queues supported by the P4000X NVMe used in our experiments, our latency measurements are affected by the round-robin scheduling mechanism implemented in the NVMe controller hardware. Some borrowers suffer from starvation: they are unlucky with regard to timing, ending up having to wait significantly longer than other borrowers for their commands to be executed by the NVMe. Furthermore, the NVMe device itself is only PCIe Gen3 $\times$ 4, and the simultaneous read requests from several nodes far exceed the bandwidth capacity of the device. Thus, a performance analysis is not particularly interesting with respect to evaluating our queue-sharing concept, as we end up evaluating how well the NVMe device performs instead. Nonetheless, while the small amount of data and low throughput in our tests may not be particularly useful for an application, we have shown that it is possible for a larger number of nodes in a cluster to access the same storage device simultaneously. In practice, we have successfully demonstrated a form of "MR-IOV in software."[13] Newer NVMes with higher bandwidth and lower latency, as well as support for a higher number of queues, will benefit from this kind of sharing capability.

7.4.3 NVMe-oF RDMA Comparison. NVMe-oF [56, 94] is a widely adopted standard for accessing remote NVMe devices over a network. NVMe-oF implementations are composed of two parts: a device-side "target" driver and a client-side "initiator" driver. The target driver is responsible for managing the NVMe device, setting up queue pairs and facilitating asynchronous access by allocating dedicated queue pairs for each individual initiator. I/O commands are forwarded by the initiator to the target driver, which enqueues them for the NVMe device. The NVMe-oF protocol is agnostic regarding the transport layer, allowing commands and completions to be transmitted over any kind of message-passing communication channel, and leaves the transportation of data entirely up to the network fabric.

For network fabrics that support InfiniBand RDMA, NVMe-oF can be supported with very high performance [29]. The defining feature of InfiniBand RDMA is that **InfiniBand channel adapters (HCAs)** may access application memory directly, allowing data to be be transferred directly from the application on one host to the application on another host without going through a network stack. By avoiding kernel transmission buffers, InfiniBand RDMA applications have very high throughput and low latency. Additionally, as the CPU is not involved in transmission, RDMA is completely asynchronous, and avoids blocked send and receive calls.

In regard to NVMe-oF, the target driver can provide direct access to both data and queue memory via system memory on the target host.[14] Application memory used for RDMA is registered with the InfiniBand driver in advance as so-called **memory regions (MRs)**. This allows the InfiniBand driver to pin the physical memory pages in memory, avoiding them being swapped out. Additionally, as it allows other hosts to resolve the local physical addresses of MRs, an NVMe-oF initiator driver can prepare I/O commands using *target-local* addresses. In other words, the initiator is able to use the target's MR as intermediate memory for NVMe data.

Similar to the SQ and CQ queue pairing mechanism for NVMe devices described in Section 6.2, InfiniBand also uses queue pairs of **work queues (WQs)** and **completion queues (CQs)**. HCAs support hosting WQs on device memory (similar to NVMe CMBs described in Section 7.4.1), and hosting CQs in system memory. This allows a userspace application to post work requests, such as send and receive operations, and poll for completions directly, bypassing the kernel entirely in the data path. An additional benefit is that this design maps very well onto the NVMe-oF architecture; the NVMe-oF target driver can "bind" the receive WQ to the NVMe SQ. This means that NVMe commands are already enqueued (in memory) when the target driver is notified about received commands, and the target driver may simply ring the SQ's doorbell register. Figure 36 illustrates the steps involved in reading 4 kB of data from storage using RDMA:

[13]Multi-Root I/O Virtualization, see Section 9.1.
[14]In RDMA terminology, this is known as "zero-copy," because the CPU is not involved in copying data. However, the authors argue that in the context of NVMe-oF, quite literally copying data from the NVMe to system memory on the target host, before sending it over the network, is actually not "zero-copy" at all.

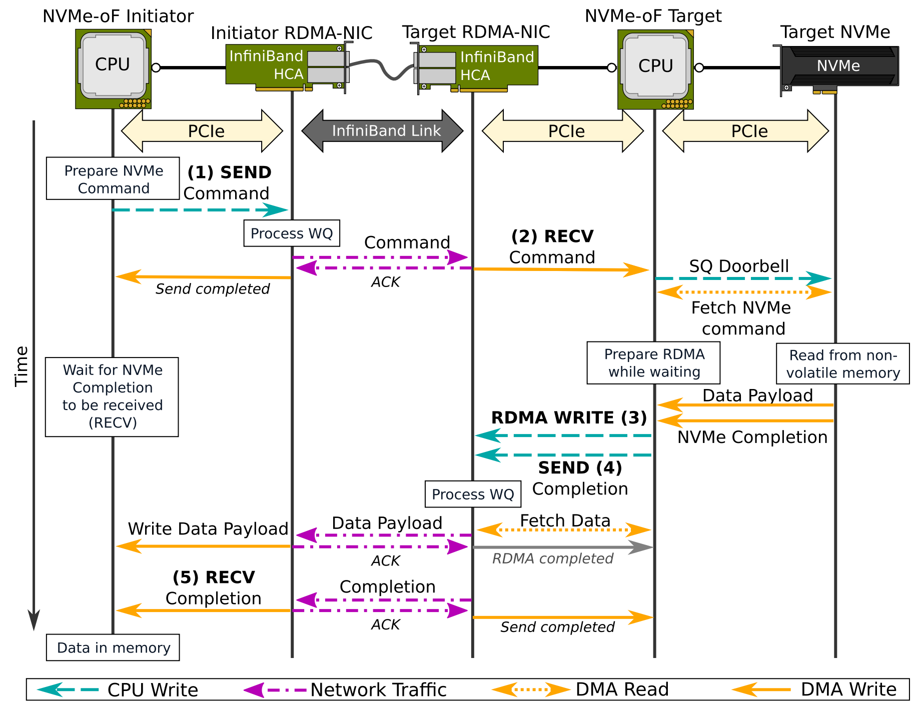


Fig. 36. Flow chart of an I/O read operation for NVMe-oF using InfiniBand RDMA. While the target-side CPU is required to initiate NVMe operations and start the RDMA write transfer, neither commands, completions, nor data is moved by the CPU. As InfiniBand queues and NVMe queues are bound to each other, commands and completions are written directly to the queues by the HCAs using DMA.

(1) The initiator prepares an I/O read command for the NVMe device with the desired block offset. Memory used for RDMA is already known to both NVMe-oF initiator, as it was registered by the target driver as a RDMA MR in advance. This allows the initiator to simply use target-side physical addresses of this MR in the read command. It then posts the command to the send WQ, sending the command across the network, directly to the target drivers memory.
(2) The target driver receives a receive completion indicating that it has received an NVMe command. As the HCA has already written the command to the appropriate location in target's memory, the target driver can immediately ring the doorbell register of the bound SQ, initiating the NVMe I/O operation. The initiator driver has already resolved target-side physical addresses in advance, so there is no processing required. After ringing the doorbell, it checks what type of NVMe command this is. Seeing that it is an read command, it starts preparing a WQ request for RDMA write from the local MR to a known MR on the initiator host.
(3) The target driver receives the NVMe command completion, indicating that the NVMe device has written data to memory. The target posts the prepared RDMA write request to the appropriate WQ. By using DMA to read from the MR, the HCA begins sending the data over the InfiniBand fabric. The initiator-side HCA will start writing received data into the initiators memory, also using DMA.

(4) Since requests in the same WQ are always ordered, the target driver immediately posts a send request for the NVMe completion, knowing that when the initiator driver receives the completion the data must have arrived before it. This optimization means that the target driver avoids needing to wait for the RDMA write completion, which is particularly useful for larger data transfers.
(5) The initiator driver receives a receive completion for the NVMe command completion, and knows that the data must have arrived in its local memory before the completion due to WQ ordering. The data read from the remote NVMe device is now available for use.

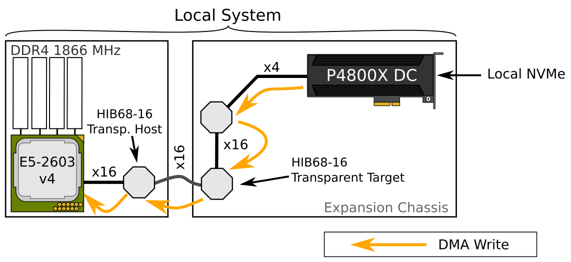
We have designed an experiment to compare the **Storage Performance Development Kit (SPDK)** [94] to our SmartIO NVMe driver implementation. SPDK is a storage application framework that implements support for a wide variety of storage devices, including NVMe devices. Similarly to our SmartIO NVMe driver, it is implemented in userspace, bypassing the kernel and primarily relying on polling. Furthermore, SPDK has a built-in NVMe-oF stack with support for InfiniBand RDMA. As such, SPDK is a suitable comparison for our SmartIO NVMe driver.

However, as SPDK and our proof-of-concept NVMe driver are two different NVMe driver implementations, comparing them to each other would be comparing apples to oranges. As such, we have instead conducted two separate tests, one where we compare the standard SPDK NVMe driver to SPDK NVMe-oF, and the other where we compare our own NVMe driver using a local and a remote NVMe device. Figure 37 depicts the four scenarios were used in our experiment:

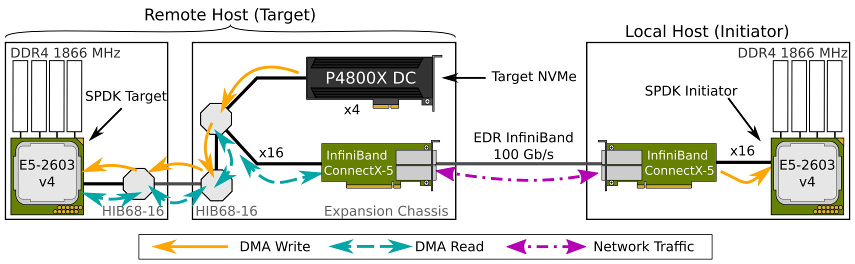
- **Local SPDK**, shown in Figure 37(a): The standard SPDK NVMe driver operating a local Intel Optane P4800X DC. The NVMe is installed in an expansion chassis, and connected upstream using the HIB68-16 transparent adapters. This scenario serves as our *local baseline* comparison for SPDK.
- **SPDK NVMe-oF**, shown in Figure 37(b): The SPDK NVMe-oF driver stack (initiator and target) operating a remote P4800X using RDMA for transport. The two hosts are connected back-to-back with two Mellanox InfiniBand ConnectX-5 EDR channel adapters. The target driver has the same CPU core affinity as its InfiniBand HCA and the NVMe device. The InfiniBand maximum transfer unit was configured to 64 kB, leaving more than enough space within a packet for the data payload. This scenario is compared to the Local SPDK scenario. Note that while we are measuring latency, the EDR speed of 100 Gb/s is equivalent to 12.5 GB/s regardless. This is similar to an x16 Gen3 PCIe link.
- **Local SmartIO**, shown in Figure 37(a): Our proof-of-concept NVMe driver implemented with the SmartIO API extension (as explained in Section 6.2), operating a local P4800X. The topology is identical as the Local SPDK scenario, but we run our NVMe driver implementation instead of SPDK. As before, the HIB68-16 transparent adapters connecting the expansion chassis use the same PEX8733 switch chips used in the PXH830 NTB adapters. This scenario therefore serves as the *local baseline* comparison for SmartIO.
- **Remote SmartIO**, shown in Figure 37(c): Our driver operating a remote P4800X NVMe. The two systems are connected back-to-back using PXH830 NTB adapters. Note that because we use the expansion chassis in our configuration, there is the same number of switch chips in the path as the Local SmartIO scenario.

On both hosts, we installed Ubuntu 18.04.2 with the 4.15 version of the Linux kernel, and we used version 19.1.1 of SPDK. We also disabled the IOMMU on both hosts in all four of the evaluated scenarios.

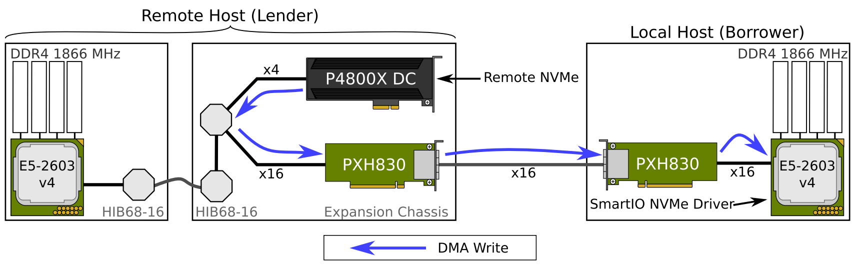
To measure read latency, we used FIO version 3.13 [9] to perform 327,680 reads, each read page-sized chunk (4 kB) with an offset generated by a pseudo-random number generator. Figure 38



(a) **Local SmartIO** and **Local SPDK**: a local NVMe device in an expansion chassis being operated by a local NVMe driver. This topology is identical for *both* SPDK and using our proof-of-concept NVMe driver. We have highlighted the direction of data being written by the NVMe to RAM.



(b) **SPDK NVMe-oF**: a remote NVMe device being operated by the SPDK NVMe-oF stack. Data must first be written to target's RAM before it can be transfered to memory on the initiator using RDMA.



(c) **Remote SmartIO**: accessing a remote NVMe device using our proof-of-concept NVMe driver. As our SmartIO sets up memory mappings for the NVMe device, it can write across the NTB using native DMA.

Fig. 37. The different scenarios in our NVMe-oF comparison experiment. Note that the local scenario is the same for both SPDK and SmartIO, the difference is only which NVMe driver software is running.

shows the latency distributions for SPDK (left) and our NVMe driver (right). We observe that compared to local access, where the NVMe device is able to access host memory directly, NVMe-oF introduces a significant performance overhead, even when using RDMA. There are two primary reasons for this performance difference. First, the CPU on the target host is involved in the critical path, as software is needed to ring the NVMe doorbell registers as well as starting RDMA writes back to the initiator. Second, to use RDMA, data must first be written to target's memory by the NVMe, in order for the InfiniBand HCA to access it and transfer it over the network fabric. In comparison, our SmartIO NVMe driver is able to initiate DMA regardless of whether the NVMe device is local or remote. Not only does this avoid the lender's CPU in the critical path entirely, but we also do not need to bounce data via memory on the lender in the same way RDMA does. In the SmartIO scenarios, because the device and the driver are the same number of switch chips

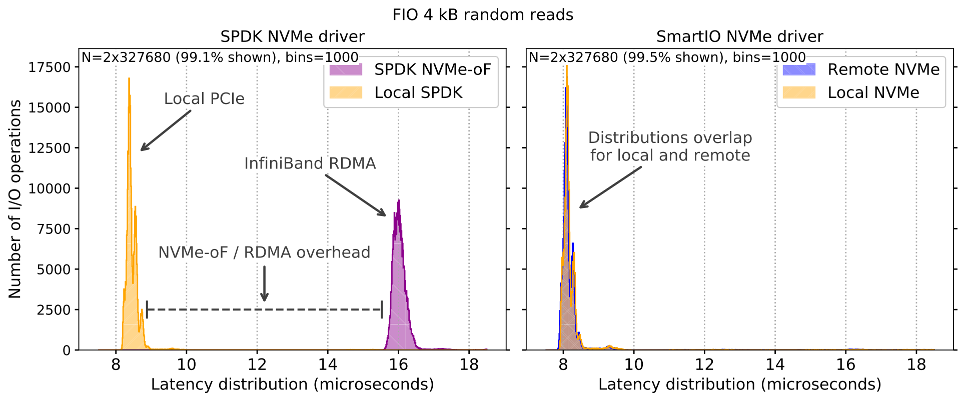


Fig. 38. Distribution of I/O command completion latencies for Local SPDK and SPDK NVMe-oF (left) and using our proof-of-concept SmartIO NVMe driver (right). By avoiding the device-side CPU in the critical path, as well as being able to use DMA directly, our NVMe driver achieves the same performance for both local and remote. Meanwhile, SPDK NVMe-oF introduces a visible latency overhead compared to local SPDK.

apart, there is *no* difference in performance for local and remote access. While SPDK and our own proof-of-concept driver are two widely different NVMe driver implementations, it is interesting to note that our driver appears to be slightly faster than local SPDK (around 600 ns on average), even for remote access.

Finally, it should be mentioned that Mellanox has implemented support for NVMe-oF target offloading in their InfiniBand adapters. Target offloading is a mechanism for avoiding target-side CPU in the critical path, by moving some of the target driver logic into hardware on the target-side HCA instead. For example, rather than relying on the target driver running on the CPU, the HCA itself can ring the NVMe doorbell by using peer-to-peer DMA when it receives an NVMe-oF command. However, we argue that a performance overhead compared to local access is unavoidable, since the RDMA mechanism inevitably requires the NVMe device to write data to memory before it can be accessed by the HCA and sent over the network.

8 DISCUSSION

Our SmartIO solution offers several benefits over traditional approaches to distributed I/O. In the previous section, we presented experiments demonstrating the usefulness and the performance benefits of SmartIO. Particularly, we have performed experiments demonstrating that it is possible to facilitate remote access to devices with native PCIe performance. In this section, we provide a short discussion on some topics and considerations that have not yet been covered by our evaluation.

8.1 Security

The challenge with security for distributed I/O and so-called “one-sided communication,” where only the initiator-side (sender) software is involved in initiating I/O but not the target (receiver), is an understudied research topic [85]. In the case of accessing remote devices using our SmartIO system, particularly DMA is a security concern. By lending away a local device, the lender effectively yields control over it to software running on a remote system. A flawed device driver on the borrower may cause a device to read from or write to rogue memory addresses on the lender. For Device Lending, it is possible to protect against unintentional memory accesses by using the lender-side IOMMU. Our SmartIO system is able to isolate devices on the lender, protecting against accidental memory reads and writes. However, the current implementation is not able to sufficiently protect against a *malicious* device driver, as any software running in kernel space on the borrower system in practice has full access to the local NTB adapter. Regardless, we argue that the in the case of a malicious kernel space driver, the entire *local* system is compromised as well. In other words, we consider this scenario to be beyond the scope of our SmartIO implementation.

In case of the SmartIO extension to the SISCI API, where we expose device driver capabilities to userspace software, a malicious program on the borrower is also a valid concern. An attacker might intentionally use a DMA-capable device to overwrite memory on the lender, causing it to crash, or use the DMA engine to snoop data from memory. In cases where the userspace software cannot be trusted, we can also use the lender-side IOMMU to protect against undesired memory accesses. By placing devices in separate IOMMU domains, SmartIO creates a virtual I/O address space per device.[15] This guarantees that the device is only able to access specific DMA windows mapped for it, thus protecting system memory and other devices on the lender. Unlike a device driver, a userspace application cannot exploit kernel space privileges to manipulate the local NTB, and is only able to set up mappings to remote memory by using the SISCI API. We argue that this provides sufficient protection against both defect and malicious userspace programs, as SISCI prevents setting up mappings to arbitrary memory by only allowing registered memory segments.

Finally, for VM pass-through with KVM, our MDEV implementation requires using the lender's IOMMU, as explained in Section 5.2. By mapping a device to the guest-physical memory layout, we limit the passed through device to only accessing DMA windows to the VM it is assigned to. In other words, it is not possible for guest software to misuse our SmartIO system to break out of the virtualized environment, since SmartIO provides the same level of memory isolation as standard pass-through.

It should be noted that relying on the lender-side IOMMU in combination with long PCIe paths may severely impair DMA performance, as we saw in our IOMMU evaluation in Section 7.3.1. As a general advice, we recommend trying to minimize the distance between a device its lender's IOMMU. Devices that support PCIe ATS [60] are able to cache resolved I/O virtual addresses, thus avoiding routing transactions via the CPU. However, it has been demonstrated that some devices, such as FPGAs and programmable network adapters, can be exploited by an attacker to abuse ATS to break out of IOMMU isolation [47].

8.2 Supported OSes

As explained in Section 4.1, PCIe devices are represented in the Linux kernel using generic device handles. This handle provides device drivers with a unified interface for accessing a device's configuration space as well as mapping DMA buffers. Through hot-adding a virtual "shadow" device handle into the Linux device tree, the borrower component of our Device Lending mechanism is able to to intercept configuration cycles and calls to the Linux DMA API. As such, we are able to inject I/O addresses that map over the device-side NTB in a manner that is transparent to the device driver.

Other OSes may represent devices differently in their system. Microsoft Windows, for example, does not provide such a unified device handle interface, and uses separate driver frameworks for different classes of devices instead. The lack of a generic PCIe device interface that we can easily manipulate makes porting the Device Lending mechanism to Windows non-trivial, and a large engineering effort is required to support similar capabilities.

[15]Some IOMMUs support isolation *per application* by using Protected Address Space ID, but as this also requires support in devices, our implementation does not currently support this.

However, supporting the lender component of our SmartIO system is more straight forward. The lender's responsibility is essentially to facilitate remote access by setting up mappings over the NTB when it is requested by a borrower. The low-level NTB driver and SISCI API are supported on a wide variety of OSes, including Windows, meaning that a Windows machine lending out its devices is possible. Additionally, as the SISCI shared-memory API is also supported on Windows, so is our SmartIO API extension. This means that while Device Lending may not be possible on Windows, implementing userspace drivers is. We have proved this by running our proof-of-concept NVMe driver on a Windows 10 installation.

Finally, it should be noted that by using our MDEV extension to KVM, devices may be passed through to a VM running *any* guest OS. By passing through an Intel Optane 900P NVMe and an Nvidia GTX 1080 Ti GPU to a VM instance using Qemu, booting the Windows 10 image from the NVMe device itself and using the GPU for video output, we have confirmed that it works. Investigating the possibility for extending our SmartIO solution by implementing support for other hypervisors, such as Xen or Hyper-V, is, however, a candidate for future work.

8.3 Supported CPU Architectures

While we primarily used Intel Xeon CPUs in our performance evaluation presented in Section 7, our implementation is not bound to any specific CPU architecture. For example, we have confirmed that our proof-of-concept NVMe driver works on an Nvidia Jetson TX2, running on its ARM Cortex-A57 processor, and accessing a remote NVMe device. Even so, our SmartIO implementation does require some considerations in regard to CPU architecture:

- Lenders must be able to support PCIe peer-to-peer to route transactions between the NTB and the device. In our experience, most CPU architectures are capable of this, but some consumer-level CPUs are not. However, this CPU limitation can be alleviated by using peer-to-peer capable switches in the PCIe tree, for example by using an expansion chassis.
- Our implementation of Device Lending only includes support for Intel and AMD IOMMUs. While the borrower's IOMMU is not strictly required for Device Lending, without it, a lender needs to map the entire memory of the borrower for the device. This limits the number of devices that can be lent out to different borrowers at the same time, as explained in Section 4.3. However, userspace drivers using our SmartIO API extension do not need the IOMMU for anything else than protecting memory. It should be mentioned that we are currently working on implementing support for IOMMU on ARM (known as the System Memory Management Unit).
- Some systems do not support assigning 64-bit I/O addresses to BARs, limiting how large the NTB BAR size can be as the combined device memory requirements must fit below 4 GB. This may limit how many devices the system is able to borrow, or how many devices the system can lend out, depending on whether the system is used as a lender or a borrower. In our experience, most modern systems support 64-bit I/O address space by enabling it in the system's BIOS.

8.4 Supported Devices

The main benefit of building our system on top of standard PCIe, is that our sharing idea will work for any standard PCIe device. As PCIe is the industry standard for connecting I/O devices to a computer system, our SmartIO system can support a wide range of devices with different form factors and connectors. Even though we primarily presented performance measurements using an Ethernet adapter, NVMe devices and Nvidia GPUs in our evaluation (Section 7), we have during the development of SmartIO experimented with FPGAs, AMD GPUs, InfiniBand HCAs, sound cards, and PCIe-attached cameras. We are even able to lend out individual functions of multi-function and SR-IOV devices to different borrowers, as shown in Section 7.1.6.

Legacy device interrupts is currently only supported by our MDEV extension to KVM. By setting up an interrupt handler on the lender for legacy interrupts, similar to how we forward interrupts in our MDEV implementation, it would be possible to use software for forwarding legacy interrupts while mapping MSI/MSI-X interrupts directly over the NTB as our current Device Lending implementation does. The same solution could be used to map MSI/MSI-X interrupts directly over the NTB for our MDEV implementation. However, we do not consider this a priority as the PCIe standard require devices to implement either MSI or MSI-X (or both) [61].

8.5 Alternative NTB Implementations

Our low-level NTB driver is not limited to the specific Dolphin NTB adapter cards and cluster switches used in our experiments, but supports multiple families of NTB-capable switch chips from both Broadcom and Microsemi. Any hardware implementation integrating one of these switch chips can be trivially supported by our driver, requiring only minor software adjustments. Additionally, the SmartIO concepts themselves are general and could be implemented for *any* NTB solution that supports similar capabilities. However, special attention may be required when using an integrated NTB as an embedded CPU rather than as a peripheral device. For example, it is possible that the lender IOMMU must always be enabled, to properly route DMA transactions over the NTB. We have not tested this, and we will investigate how embedded NTBs can be supported in future work.

Although the SmartIO implementation is incorporated into Dolphin's software stack due to its high-level shared memory support, it should be mentioned that the Linux kernel also has an NTB driver framework [35]. A handful of NTB implementations are already supported in the kernel through this framework, such as Microsemi switches, Intel Xeon's NTB, and the AMD Zeppelin NTB. While this framework has only rudimentary support for low-level NTB functionality, i.e., setting up memory mappings and configuring interrupts, we hope that NTBs' potential for PCIe-based interconnection and shared-memory clustering is something that eventually may be picked up by the community.

8.6 Scalability

The Dolphin PXH830 NTB adapters and MXS824 cluster switches used in our experiments support external copper cables of lengths from 0.5 m up to 5 m. It is also possible to use fiber-optic cables that can be up to 100 m long [21]. The MXS824 cluster switch has 24 × 4 Gen3 ports, which can be configured to different combinations of ×4, ×8, and ×16 links. By connecting 7 switches (1 top switch and 6 subswitches) in cascade, and connecting each node to a subswitch through a x8 link, we demonstrated in Section 7.4.2 that our SmartIO solution works in a 60 node cluster sharing an NVMe device. However, while up to 60 nodes can be supported in the cluster, there are some limitations with regard to the number of *devices* that can be supported.

One such limitation is the number of available look-up table entries in the NTB implementation. As we briefly discussed in Section 3.3, the number of mappings over the NTB is limited by the number of entries in the NTB's internal look-up table. Reading from remote memory is a non-posted request that require a completion, as we described in Sections 7.1.3 and 7.3.1. While the request is routed based on its *address*, the completion (with data) is routed back again based on the *requester*. This means that to support read operations to remote memory, the NTB must support mapping requesters as well as addresses, to make sure that completions are routed back to requesters through the NTB. In other words, NTBs have two kinds of look-up tables, one used for translating a local I/O address into an arbitrary remote address, and another used for returning completions to the appropriate requester (a CPU or a device). The number of devices that can be borrowed or lent out is limited by the size of this requester look-up table.

The PXH830 NTB adapters used in our evaluation support 32 such requester mappings per adapter card. With two nodes connected back to back with PXH830 adapters, each of the two nodes may borrow up to 30 devices from the other node and (simultaneously) lend out up to 30 local devices. In this context, we are referring to devices, rather than individual *device functions*, and any of these 30 devices may have several device functions (such as an SR-IOV-capable device). Two mappings are reserved for each of the CPUs, which must also be able to reach across the NTB due to our implementation of the underlying shared-memory communication. While any single node may only lend out 30 local devices and/or borrow up to 30 remote devices, it is possible to add switches to the topology and connect more nodes, thus increasing the total number of available devices in the cluster.

However, the cluster switch itself also has a finite number of available requester mappings per NTB-capable switch port. Setting up an outgoing requester mapping on one switch port consumes ingoing requester mappings on all the other ports. Therefore, adding switches and nodes to the topology will consume requester mappings cluster-wide, as CPUs will require two requester mappings each to reach all the other nodes in the cluster. Although the number of these mappings is very high, it does not scale indefinitely. The exact threshold for when adding more nodes starts decreasing the possible number of devices that can be shared varies, and depends on the configuration of the cluster. However, this limitation can be avoided by designing the cluster topology with device sharing in mind, rather than maximizing the number of nodes.

Another limitation on the number of devices a borrower is able to borrow is the NTB BAR size, or, the size of the "NTB window." As the borrower must map device BARs through its local NTB, borrowing devices with large BARs would use up more of the NTB window than devices with smaller BARs. For example, it would most likely be possible to borrow more NVMes than GPUs, as NVMes usually have smaller device memory requirements than GPUs. Moreover, the NTB window size can also affect how many devices a lender may lend out at any given time. Devices that require large DMA windows would use more of the NTB window than devices that do not require large DMA transfers. Because of this, it is desirable to set the NTB window size as large as possible.

However, some devices may have addressing limitations making them incapable of reaching high memory addresses. This can become an issue in the case where a lender has many devices or where the workflow requires very large DMA windows, and we need to configure a very large NTB BAR size. As we explained in Section 4.3, increasing device memory requirements may force the system to place the NTB at a high address. The sum of all device memory requirements, i.e., the combined size of the combined downstream BARs (including the NTB), may be so large that the system is forced to assign device memory at high addresses. In the case of the NTBs in our evaluation, devices with addressing limitations would be incapable of reaching DMA windows. The lender-side IOMMU can be used to remap DMA windows from high to low addresses for devices with addressing limitations, but this may come with a performance cost as we observed in Section 7.3.1. Without the lender's IOMMU, the number of devices within a lender is therefore limited by the devices' memory requirements and addressing capabilities. However, in cases where device memory limitations is a concern, it is possible to simply add more dedicated lender nodes to the topology. This way, we can spread out devices over several lenders, ensuring that that any one lender's combined device memory requirements does not exceed the system's low/high memory assignment threshold.

Thus, the limitation on the number of devices and nodes depends on several factors, such as cluster topology, addressing capabilities of the devices, memory requirements of the devices, and the NTBs' look-up table sizes. As such, there is no straight forward answer to the question of how many devices and nodes can be supported (beyond the topologies already described in this article). However, it should be noted that these limitations stem from limitations in the hardware implementations of different devices and NTBs, and not from our SmartIO sharing methods.

8.7 Disaggregated and Composable Infrastructure

Using SmartIO, it is possible to design custom configurations of remote and local devices on the fly, while all systems are running. Multiple hosts in the PCIe-networked cluster may contribute their devices, effectively turning the entire cluster into a shared, disaggregated PCIe infrastructure. Individual nodes can dynamically allocate device resources according to immediate application requirements, and release them when they are no longer required. This can potentially greatly increase the utilization of devices in the cluster, as devices are no longer tightly coupled with the hosts they are installed in.

Moreover, as it is possible to use all three sharing aspects of our SmartIO framework, i.e., Device Lending, MDEV, and the new SmartIO API extension, in different combinations, we are able to support a wide variety of applications at different abstraction levels. Our system effectively eliminates the distinction between local and remote, as well as device and system memory, providing great flexibility with regard to heterogeneous cluster computing. This makes it easier for an application developer to scale out in a cluster and design advanced cluster workflows, e.g.:

- Using Device Lending, remote devices appear to a system as if they are locally installed, facilitating remote access in a manner that is completely transparent to device driver, application, and even the OS. Large-scale CUDA programming tasks can make use of multiple GPUs that appear to be local, instead of writing a distributed applications or relying on middleware such as rCUDA [23, 68]. In Section 7.2, we for example demonstrated that it is possible to scale-out a GPU-intensive convolutional neural network training task. Pogorelov et al. [66] have previously shown how a multimedia workload can be offloaded to remote GPUs using Device Lending to meet real-time requirements.
- Access latency in NVMe-oF can be avoided by borrowing the remote NVMe device instead, and accessing it directly, either by using Device Lending as demonstrated in Section 7.1.1 or extending our proof-of-concept NVMe driver as demonstrated in Section 7.4.3. Distributed database applications may reduce query times by using a combination of local and remote NVMe devices for caching and data consistency. By distributing I/O queues for NVMe devices to multiple nodes as demonstrated in Section 7.4.2, it becomes possible for each node to control data locality and thereby reduce the latency for data consistency across nodes.
- Using Device Lending, high-speed network interfaces, such as InfiniBand HCAs and 100 Gigabit Ethernet adapters, can be assigned to a node while it needs high throughput, and released when no longer needed. Furthermore, many network interfaces are also capable of SR-IOV, allowing a single network card to be distributed to multiple cluster nodes simultaneously, without requiring any software adaptions as demonstrated in Section 7.1.6.
- In addition to enabling access to individual remote devices, SmartIO also supports creating groups of arbitrary devices and enabling direct peer-to-peer access between them. This makes it possible to create workflows that are optimized for both resource utilization and data locality. By combining Device Lending and the SmartIO API extension, we demonstrated in Section 7.4.1 how it is possible to stream data directly into GPUDirect-capable GPUs across the PCIe network. In Section 6.4, we also explained how a long-running GPU kernel may load and store data by itself, eliminating CPUs and system RAM in the data path entirely.

Throughout this article, we have demonstrated how we can facilitate remote access to devices, without any performance overhead compared to local access. As such, we argue that we have demonstrated that our SmartIO sharing system turns a PCIe shared-memory cluster into a distributed, composable infrastructure.

9 RELATED WORK

As a complete system with several components, each component of SmartIO could potentially be discussed at great length to place them in proper context. In fact, several aspects of related work has already been presented throughout the article, such as PCIe shared-memory networking in Section 3 and an implementation of NVMe-oF using RDMA in Section 7.4.3. Our SmartIO solution is at its core a system for sharing I/O devices and facilitating remote access. We have therefore condensed this section to compare our solution to disaggregation solutions we consider the most relevant. In particular, we summarize disaggregation techniques based on PCIe fabric partitioning, followed by a comparison to I/O distribution solutions implemented with NTBs. We also provide a short discussion on using RDMA for distributed I/O. This is followed by some background for the ideas behind our proof-of-concept NVMe driver, before we finally present memory disaggregation ideas that are related to our shared-memory techniques.

9.1 PCIe Fabric Partitioning

The idea of using the PCIe bus as a shared interconnection fabric between independent host systems is not new. An early approach is **Multi-Root I/O Virtualization (MR-IOV)** [59]. MR-IOV specifies how a single PCIe fabric may be logically partitioned into separate virtual PCIe trees, where each host sees its own PCIe tree without knowing about the others. This partitioning becomes possible using special multi-root aware switches in the fabric. Additionally, in the same way SR-IOV requires virtualization support implemented in hardware, MR-IOV too require devices to be multi-root aware and support multi-host access. Devices without multi-root capabilities can not be shared on the function level. Due to the complexity of implementing MR-IOV, particularly its requirement that both switches and devices are multi-root ware, it did not see any widespread adoption. At the time of writing, we are not aware of any commercially available devices capable of MR-IOV. Wong [92] have demonstrated live partitioning using PLX/Broadcom switches without requiring multi-root aware switches and devices, but their solution does not allow splitting individual device functions or simultaneous access from multiple CPUs either.

Rack-scale computers [17, 18] are computer systems where multiple (independent) CPUs and free-standing I/O devices are attached to the same PCIe fabric, usually connected by a so-called "top-of-rack" PCIe switch. These solutions support disaggregation by dynamically partitioning the shared fabric into different "subfabrics." The partitioning is made possible by prefixing standard PCIe transactions with a custom header extension called fabric IDs. Devices are transparently attached to their respective partitioned fabric, and are only seen by a single CPU at the time. Unlike MR-IOV, these partitioning solutions does not require support in devices, but they do require dedicated switch chips that support the proprietary fabric ID header extension to configure routes between devices and CPUs through the fabric. Chung et al. [15] present a proprietary solution using Broadcom PEX9797 chips to partition the fabric and distribute individual SR-IOV functions. Similar solutions also exist for Microsemi switch chips [51].

Solutions based on partitioning allow devices to be disaggregated at the (virtual) function level, thus they can be said to enable a composable infrastructure. However, they do not specify any memory-to-memory communication between hosts. In other words, partitioning solutions do not offer any shared-memory capabilities as part of the system, making a solution like our device driver API extension impossible. Consequently, fabric partitioning does not provide the same level of sharing capabilities compared to our shared memory-based system, and simultaneously sharing a device between multiple CPUs requires additional distribution methods, such as RDMA. In contrast, our SmartIO implementation is not only able to share distribute individual device functions (both physical functions and SR-IOV virtual functions), but makes it possible to implement "MR-IOV in software" even for non-SR-IOV single function devices. Our proof-of-concept NVMe driver described in Section 6.2 demonstrates this in practice.

It should also be mentioned that most solutions based on fabric partitioning are only modular to the extent of a typical blade server configuration, and scaling beyond this requires I/O distribution over traditional network. As many of them rely on proprietary technology, adding new I/O devices or CPUs to the configuration requires additional modules, often only available from the same vendor. In comparison, SmartIO is fully distributed, and enables a heterogeneous computing system, where different CPU architectures may be connected in a cluster and sharing their devices. Any standard PCIe device may be distributed and shared.

9.2 NTB-based Solutions

Using the same Broadcom PEX8733 switch chips used in Dolphin's PXH830 NTB adapters, Lim et al. [44, 75] have developed NTB host adapters and connected three hosts in a cluster. By extending a shared-memory API with NTB support, their focus seem to be shared-memory functionality for high-performance computing applications, and distributing devices appears not to have been considered. It should be noted that the memory-mapping capabilities they have developed for their API support are similar to functionality already existing in the SISCI API [22].

Bielski et al. [12] summarize various disaggregation solutions of I/O devices in the context of high-performance computing. Interestingly, they point out that NTB-based device distribution solutions appear to have relied almost exclusively on network adapters in their performance evaluations, as they were only able to find one example that used GPUs in their evaluation. Additionally, they also point out that most solutions for disaggregating SR-IOV devices seem to be limited to distributing virtual functions to (remote) VMs. Our SmartIO system, however, works for any standard PCIe device. We have used network adapters, NVMe devices and GPUs in our experiments presented in Section 7. Moreover, while we also support pass through to VMs using our MDEV extension (Section 5), we have demonstrated how we can share individual virtual SR-IOV functions to bare-metal machines in Section 7.1.6. Additionally, it becomes possible to disaggregate single-function devices in software by using our SISCI API extension, as demonstrated by our NVMe driver implementation explained in Section 6.

Suzuki et al. [79] have implemented NTB-like capabilities in an FPGA in order distribute SR-IOV functions to different hosts. Although their solution is specific to tunnelling PCIe over Ethernet (ExpEther), their initial performance evaluation showed promising throughput measurements for a Gen2 x8 PCIe device. The authors have since shown that the additional network latency has a negative performance impact for DMA reads [78].

Guleria et al. [27] propose to connect an expansion chassis to one or more hosts using an NTB. By using an ARM CPU add-in card that enumerates the devices in the expansion chassis, they propose an interesting solution that allows programmable devices, such as a GPU, to continue to operate independent of host assignment. However, the implementation of the actual distribution method seems to be lacking, and the authors do not suggest any solutions for allowing the device to be seen by multiple CPUs, or providing any mechanism for dynamically setting up any memory-mappings necessary for DMA. Instead, they propose various traditional distribution methods, such as RDMA or adapting GPU-specific middleware.

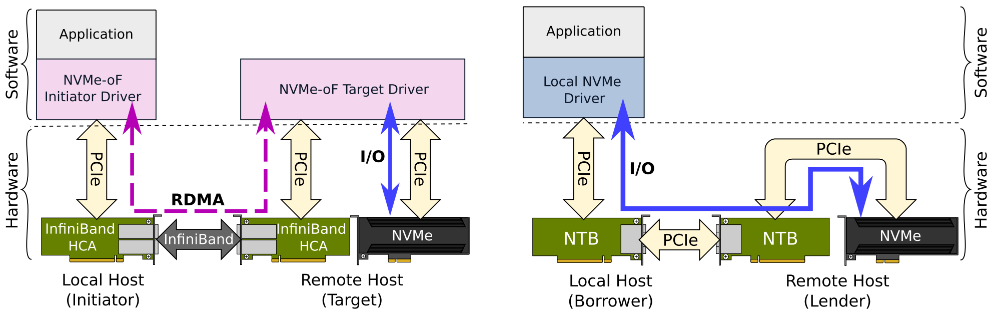
Hou et al. [31] present a solution where hosts are connected to NTB-capable ports on a Broadcom PEX 8648 switch chip. Devices are installed in a dedicated server host, which enumerates the devices and assists the other hosts in setting up NTB mappings. However, instead of leveraging the NTB to map device BARs for the local host and mapping memory for the device, their solution appears to be based on transferring data from memory to memory, and then involving the local device driver on the dedicated server. This solution incurs a performance reduction compared to local device access, as reflected in their performance evaluation.

The Ladon system [88] provides functionality that is very similar to our own MDEV implementation, and could potentially be extended to support something similar to both Device Lending and our device driver API. By using a top-of-rack switch with NTB-capabilities, Ladon facilitates access to the same SR-IOV device from multiple VM guests. The device and a dedicated "management host" are connected to the switch transparently, and the management host enumerates the PCIe fabric and takes ownership of the device. In that regard, the management host is conceptually similar to our lender. Multiple "compute hosts" are connected to the same switch through non-transparent switch ports, i.e., NTBs. The management host maps the entire memory of each compute host for the device, and assists the compute hosts in setting up mappings to individual (virtual) device functions, to pass them through to VM guests running on the compute hosts. Ladon's static setup, where all hosts are connected directly to the same top-of-rack switch as the device and the entire memory of each compute host is mapped, allows transactions to be routed directly to each compute host without relying on the IOMMU on the management host. Additionally, by extending the compute hosts' hypervisor with a specialized host driver, Ladon can support mapping MSI-X interrupts directly into VMs [86, 89]. However, by requiring device-specific drivers, this interrupt mapping does not appear to be a generic solution.

The main difference between Ladon and our SmartIO solution is that while a single host owns the device in Ladon, our SmartIO system is truly distributed by supporting multiple hosts acting as lenders. Hosts may even act as both lender and borrower at the same time. Moreover, in Ladon, the management host becomes a single point of failure. Ladon has since been extended with fail-over support, allowing a back-up management host to copy the PCIe fabric enumeration of the first host, and seamlessly take over ownership of the device in case the first management host fails [87]. However, we argue that this still does not make Ladon distributed in the same sense our SmartIO system. It is not possible for a compute host to use devices from *different* management hosts. In other words, the Ladon system appears to be limited to devices attached directly to a single rack switch managed by a single host (with fail-over). In contrast, SmartIO solution supports scaling out and using devices from several hosts across an entire cluster. Because the Ladon implementation maps the entire memory space of each physical compute host (rather than just memory used by the VMs), the number of compute hosts in Ladon setup will be limited to a handful hosts due to the combined device memory requirements of the NTBs. Our MDEV implementation, however, scales better by probing the guest-physical memory layout and only mapping VM memory, as explained in Section 5.2. Not only does this allow a lender to support more borrowers as we are able to fit more DMA windows through the NTB BAR, but we can simply add more lender systems should device memory requirements become an issue. Finally, the Ladon system appears to work only for VMs unless device-specific host drivers are implemented. Our SmartIO system, however, supports both physical machines and VMs alike by combining Device Lending and the MDEV extension. With Device Lending, devices can be used by the bare-metal host without requiring any modifications to driver software.

9.3 Distributed I/O Using RDMA

There are several widely adopted high-speed interconnection technologies used in high-performance computing clusters today, such as InfiniBand and 100/200 Gigabit Ethernet. To make



(a) NVMe-oF using RDMA (b) SmartIO

Fig. 39. Even though RDMA allows efficient memory-to-memory transfer, a device-side driver is still needed to initiate the I/O operation. Using our SmartIO solution, the local device driver can initiate I/O operations directly and avoids software in the critical path.

use of their high throughput and low latency, many cluster applications make use of RDMA [84]. RDMA variants are summarized by Huang et al. [32], and include RDMA over InfiniBand, **RDMA over Converged Ethernet (RoCE)** and **Internet Wide Area RDMA Protocol (iWARP)**. By using one-sided communication and providing direct access to application memory, RDMA solutions have been shown to greatly improve performance for a variety of distributed applications [32, 34, 45].

One of the most successful GPU disaggregation frameworks on the market today is rCUDA [23, 68]. Similarly to how the shadow device in our Device Lending mechanism makes it possible to intercept calls to the kernel's DMA API, rCUDA uses virtualization techniques to intercept CUDA API calls and enable access to remote GPUs while the programming model remains simple. By supporting GPUDirect, rCUDA and other RDMA-based GPU disaggregation solutions are able to copy data directly in and out of GPU memory using RDMA with very high performance [70, 91]. However, these solutions are not as closely integrated to the PCIe fabric as our NTB-based solution; using Device Lending or our MDEV extension, we are able to support CUDA Unified Memory [73], allowing GPUs to access host memory and memory of other GPUs directly. We are not aware of any RDMA-based GPU disaggregation solutions that are able to support this.

Many different frameworks for distributed I/O using RDMA exist, such as NVMe-oF [28, 29, 56] and rCUDA discussed above. However, RDMA solutions are tightly coupled with the device (or type of devices) they are implemented for. As illustrated in Figure 39, even though RDMA facilitates memory-to-memory transmission, a specialized device driver is still required on the device-side system to initiate the actual I/O operation.

Additional software complexity in the form of target-side drivers inevitably leads to a performance overhead compared to accessing a local device, as we observed in our NVMe-oF comparison in Section 7.4.3. Some of this target driver functionality can be implemented in network adapter hardware, for example in the case of NVMe-oF target offloading. Another approach is implementing network interface capabilities directly into device controllers, as proposed by Daglis et al. [19]. While such solutions may improve I/O performance to the point were it becomes comparable to local access, we argue that these solutions become even more coupled with the specific devices they are implemented for by requiring implementation in hardware. In contrast, our SmartIO system is general in terms of device support, as we can distribute any PCIe device without requiring specific support in devices or device drivers.

9.4 NVMe Queue Distribution

Using the SmartIO extension to the SISCI API, we have implemented a working proof-of-concept userspace NVMe driver, as described in Section 6.2. To the best of our knowledge, our driver is unique in that it is able distribute individual I/O queues of a single-function NVMe device to *remote* systems in an cluster, without using RDMA. As such, we disaggregate an NVMe device at the software-level. However, similar ideas for sharing an NVMe device at the queue-level for userspace applications running on the same *local* system can be found in several implementations, including SPDK [94].

Peng et al. [58] implement a paravirtualized NVMe driver using the same mediated device driver interface we have used for our MDEV implementation. Instead of passing through the device itself, their solution is based on using passing through I/O queues instead. They accomplish this by assigning individual I/O queues to emulated NVMe child devices. An interesting observation is that the authors report that relying on polling instead of interrupts significantly increases performance, which could suggest that their performance measurements are affected by same interrupt notification delays we observed in our own MDEV evaluation (Section 7.3.2). Furthermore, Kim et al. [39] extend the Linux NVMe driver with a dedicated queue management kernel module that is responsible for creating and deleting SQs and CQs, as well as mapping DMA buffers and doorbell registers for a userspace application. This way, a userspace application is given control over queue memory and can submit I/O commands and poll for completions directly, without going through the kernel block layer. By mapping queue memory directly for the application, this solution is conceptually very similar to how our own driver is implemented, but by using our SmartIO system we can assign queues to applications running on *remote* hosts as well.

Our NVMe driver implementations also supports GPUDirect. Although several solutions using GPUDirect to facilitate peer-to-peer access between an NVMe device and a GPU already exist [10, 11, 40, 83], we believe our proof-of-concept device driver's ability use (remote) GPU memory to host I/O queues closer to an NVMe device to be a new idea. This becomes possible by combining our driver with using Device Lending to access remote GPUs. We have demonstrated the latency benefit of this in Section 7.4.1.

Additionally, our implementation supports offloading NVMe operation onto a GPU entirely and eliminating the CPU in the data path altogether, as we explained in Section 6.4. While this would arguably prove to be highly useful in the case of a local system, the utility of this increases significantly for applications that can now freely make use of accelerators, storage devices and memory anywhere in a cluster and optimize the data flow through the PCIe network. Supporting this kind of flexibility while allowing applications and application programmers to remain agnostic about address space layout on remote systems is, to the best of our knowledge, a novel contribution.

9.5 Memory Disaggregation

In our work, we enable efficient distribution of devices across a cluster system, alleviating both load balancing problems and lack of or limited numbers of local devices. While we are primarily concerned with I/O device sharing to make active resources available to cluster nodes, our SmartIO implementation is made possible through distributed shared memory. After all, we are effectively mapping and enabling access to remote memory regions. As such, our work has an inherent relationship with memory disaggregation techniques.

Memory disaggregation concepts originally sprung out of related ideas from early work on distributed memory and distributed OSes. Since CPUs have only operated on local memory, scarce memory would be augmented by swap space. Remote memory has frequently been proposed as a faster alternative to disk-backed swap spaces [24, 26, 42, 46], to overcome the limited throughput and high latency of hard disks. Although this premise has been questioned due to the software overhead [8], performance gains have been measured with both centralized [24] and decentralized [42, 46] approaches. By relying on the same PCIe-based distributed shared memory capabilities that our own SmartIO is built on, an implementation for (partial) memory disaggregation solution can be imagined. If combined with our extended SISCI API explained in Section 6.1, then it could support a combination of local and remote RAM, as well as remote *device memory*. In fact, we have already demonstrated something similar to this in Section 7.4.1. Even though the main purpose of this experiment was to prove reduced memory access latency for a remote NVMe, it also showed that we are able to map both remote RAM and onboard device memory (of the remote GPU) for the local CPU.

More recent memory disaggregation solutions rely on RDMA for efficient access to remote memory. Gu et al. [26] show how software overhead of swapping to remote memory can nearly entirely be avoided by using RDMA. Similarly, Aguilera et al. [5] propose a solution where clients use remote memory more explicitly, through a file system-like API that acts as an abstraction over RDMA. The most interesting aspect of this idea is that as their file system interface supports POSIX semantics, it becomes possible to support the `mmap` operation. A local process may memory-map a file descriptor, and, by relying on virtual memory trapping (page faults), RDMA transfers are initiated under the hood. By using the SISCI API to map remote memory directly into a process' virtual address space, we avoid any latency from handling traps in software. Instead, the local CPU can access memory across the NTB directly, thus avoiding software in the path altogether.

So-called "byte-adressable" NVMe devices are becoming increasingly common. These NVMes implement memory controllers and expose non-volatile flash memory through device BARs, similar in concept to the GPUDirect-capable GPUs used in our experiments. As such, they are frequently used for persistent memory solutions [71, 93]. Abulila et al. [4] argue that because non-volatile flash memory is approaching dynamic RAM speeds, traditional swapping semantics incur significant system performance overhead. They propose an extension to the Linux kernel virtual memory manager that short-cuts the Linux block-layer, to support efficient swapping to byte-addressable NVMe devices. With our SmartIO API extension, this solution could be extended to *remote* NVMe devices as well, by mapping the remote BAR for the local CPU. However, additional adaptions would be required, to limit or, preferably, avoid reading over the NTB.

Although the use of dedicated blade servers may stretch the term disaggregation, Lim et al. [43] nevertheless propose an interesting solution for swapping to remote memory blades over PCIe. They suggest a hardware modification to the memory controller by which the CPU could prefetch cache lines directly over the PCIe bus and "fault in" remote memory pages, by initiating DMA transfers on the remote server. While their proposed solution would avoid reading over the PCIe bus, their evaluation appears not to take into account any latency that would be added by this hardware DMA mechanism.

The disaggregation concept is perhaps taken to its most extreme by LegoOS [74]. Here, processing, memory, and storage resources are all encapsulated into components that can be combined arbitrarily to serve cluster applications. Other hardware devices can be encapsulated in the same manner. Although the authors note that it is not possible to fully separate CPUs and memory management in practice, their idea of building disaggregation concepts into the OS itself instead of using middleware is intriguing. LegoOS only stops short of being a fully distributed OS by presenting users with virtualized nodes that appear as individual VMs. While it is possible to run existing Linux applications on these virtual nodes, device drivers must be adapted to fit this new

OS design. In contrast, our own Device Lending mechanism is able to facilitate this kind of disaggregation at a level "underneath" the OS. In effect, it is possible to use remote devices without requiring any modifications to existing device drivers. Finally, LegoOS claims to have performance comparable to a standard Linux server, but their NVMe benchmark shows a significant reduction in number of I/O operations per second compared to Linux when the amount of data is more than a few kilobytes. This performance gap is explained with network overhead. In comparison, because our own SmartIO solution is PCIe-based, we have *zero* overhead compared to local access, as shown in Section 7.

10 CONCLUSION

In this article, we have presented our SmartIO system for efficient, zero-overhead sharing of I/O devices in a heterogeneous PCIe cluster. By using memory-mapping capabilities inherent in NTBs, we combine traditional I/O with distributed shared-memory functionality over native PCIe. Our system consists of the following five components:

(1) Our low-level NTB driver, facilitating shared-memory abilities and providing mechanisms for mapping remote memory. As such, we use this to create a global address space comprised of all hosts in the cluster, including their internal devices and memory.
(2) A common abstraction mechanism, providing the necessary functionality for translating remote I/O addresses and resolving paths in the network. This allows both software and devices to be agnostic about address space layouts in remote hosts.
(3) Our Device Lending method, making remote devices appear to a system as if they are locally installed. Existing device drivers, application software, and even the OS itself may use the remote devices without requiring *any* adaptions.
(4) Our MDEV extension to KVM hypervisor, allowing pass-through of remote devices to a VM, and enabling direct access to hardware over native PCIe without breaking out of memory isolation.
(5) Our new SmartIO device driver API extension to the SISCI shared-memory API, enabling cluster applications to take full advantage of shared-memory capabilities and write device drivers optimized for shared-memory cluster workloads.

Additionally, we have also presented our proof-of-concept NVMe driver implementation, using our SmartIO API extension. This driver demonstrates several aspects of I/O with shared-memory capabilities, such as simultaneously sharing a non-SR-IOV device among multiple hosts ("MR-IOV in software") and enabling peer-to-peer memory access to (remote) GPUDirect-capable GPUs.

Using our SmartIO system, devices can be distributed in a way that meets current processing requirements, while at the same time the overall resource utilization in the cluster is improved as resources are no longer locked to individual hosts. We prove the flexibility of our solution through a broad range of performance evaluations for different scenarios and topologies for all three distribution aspects of our SmartIO system, i.e., Device Lending, MDEV, and the API extension (in the form of experiments with our proof-of-concept NVMe driver). By comparing the performance of using remote devices to local access, our results show that we do not add *any* performance overhead beyond what is expected for longer PCIe paths.

While our current system shows great potentials for resource sharing, there are still several areas that still may be investigated. For example, currently only cold migration of VMs with passed-through devices is possible. Adding support for a dynamic migration during runtime (live migration) is desirable. Such a solution would most likely require the development of new hardware as it must support either pausing or re-routing transactions without violating strict ordering required by PCIe. Additionally, our experimental results also show that a major performance bottleneck occurs when DMA read requests traverse the lender's CPU in order for addresses to be resolved by the IOMMU. The Intel Xeon CPUs used in our performance experiments alter the requests in a way that leads to poor link utilization. As our MDEV implementation requires the lender's IOMMU to map guest-physical memory for the device, this warrants further evaluations of alternative CPU architectures. Furthermore, while our proof-of-concept NVMe driver provides block-level access for userspace applications, implementing a file system or coordinating access is currently the responsibility of the application. Another candidate for improvement is therefore to implement the sharing idea in a kernel space driver, making it possible to implement a shared-disk file system for general use. As the device is shared on the queue-level, this solution could easily co-exist with the existing userspace implementation, and we could assign queues to both application instances and kernel drivers alike. Finally, as the Intel P4800X NVMe used in our queue-sharing experiments did not perform adequately, it would prove useful to perform a large-scale evaluation of our queue-sharing concept using a newer PCIe Gen4 NVMe with greater bandwidth capacity and support for a higher number of queues.

AVAILABILITY

The source code of our proof-of-concept distributed NVMe driver is licensed using the BSD software license, and is available at the following URL: https://github.com/enfiskutensykkel/ssd-gpu-dma/.

The source code of the ping-pong CUDA program used in our latency evaluation can be found at the following URL: https://gist.github.com/enfiskutensykkel/2b0f7afcb35d12477165746f062c7453.

The datasets and benchmarking results in this article are available from the corresponding author upon request.

ACKNOWLEDGMENTS

The authors thank the Dolphin Interconnect Solutions developers team, particularly Eivind Bergem and Eivind Eriksen for their input on data visualization. We also thank Friedrich Seifert, Preben N. Olsen, and Calin Iaru for their feedback on the manuscript. The authors also thank Hugo Kohmann and Roy Nordstrøm. Finally, we give a big thank you to all the anonymous reviewers. They have had a tedious task reviewing this long manuscript but still have provided a list of valuable insights and suggestions that greatly improved this article.

REFERENCES

[1] Keras. [n.d.]. Retrieved from https://keras.io.

[2] TensorFlow. [n.d.]. Large-Scale Machine Learning on Heterogeneous Systems. Retrieved from https://www.tensorflow.org/.

[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 Technol. J.* 10, 03 (2006). https://doi.org/10.1535/itj.1003.02

[4] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 971–985. https://doi.org/10.1145/3297858.3304061

[5] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: A simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*. 775–787.

[6] Knut Alnæs, Ernst H. Kristiansen, David B. Gustavson, and David V. James. 1990. Scalable coherent interface. In *Proceedings of the International Conference on Computer Systems and Software Engineering (CompEuro'90)*. 446–453. https://doi.org/10.1109/CMPEUR.1990.113656

[7] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. Springer, 256–274. https://doi.org/10.1007/978-3-642-24322-6_22

[8] Eric A. Anderson and Jeanna M. Neefe. 1994. *An Exploration of Network RAM*. Technical Report. EECS Department, University of California. Retrieved from https://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/CSD-98-1000.pdf.

[9] Jens Axboe. [n.d.]. Flexible I/O Tester. Retrieved from https://github.com/axboe/fio.

[10] Stephen Bates. 2015. Project Donard. Retrieved from https://github.com/sbates130272/donard.

[11] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*. 665–676.

[12] Maciej Bielski, Christian Pinto, Daniel Raho, and Renaud Pacalet. 2016. Survey on memory and devices disaggregation solutions for HPC systems. In *Proceedings of the International Conference on Computational Science and Engineering and International Conference on Embedded and Ubiquitous Computing and International Symposium on Distributed Computing and Applications for Business Engineering (CSE-EUC-DCABES'16)*. 197–204. https://doi.org/10.1109/CSE-EUC-DCABES.2016.185

[13] Broadcom. 2011. PEX8733, PCI Express Gen 3 Switch, 32 Lanes, 18 Ports. Retrieved from https://docs.broadcom.com/docs/12351852.

[14] Broadcom. 2012. PEX8796, PCI Express Gen 3 Switch, 96 Lanes, 24 Ports. Retrieved from https://docs.broadcom.com/docs/12351860.

[15] I.-Hsin Chung, Bulent Abali, and Paul Crumley. 2018. Towards a composable computer system. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia'18)*. 137–147. https://doi.org/10.1145/3149457.3149466

[16] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. 2013. Deep learning with COTS HPC systems. In *Proceedings of the International Conference on Machine Learning (ICML'13)*. 1337–1345.

[17] Intel Corporation. 2015. Intel Rack Scale Design. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[18] Liqid Corporation. [n.d.]. Liqid Composable Infrastructure. Retrieved from https://www.liqid.com/.

[19] Alexandros Daglis, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2015. Manycore network interfaces for in-memory rack-scale computing. *ACM SIGARCH Comput. Architect. News* 43, 3 (2015), 567–579. https://doi.org/10.1145/2872887.2750415

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR'09)*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[21] Dolphin Interconnect Solutions. [n.d.]. SFF-8644 MiniSAS-HD PCIe Gen3 cables. Retrieved from https://www.dolphinics.com/products/PCI_Express_SFF-8644_cables.html.

[22] Dolphin Interconnect Solutions [n.d.]. *SISCI API Documentation*. Dolphin Interconnect Solutions. Retrieved from http://ww.dolphinics.no/download/SISCI_DOC_V2/.

[23] José Duato, Antonio J. Pena, Frederico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS'10)*. 224–231. https://doi.org/10.1109/HPCS.2010.5547126

[24] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. 1995. Implementing global memory management in a workstation cluster. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'95)*. 201–212. https://doi.org/10.1145/224056.224072

[25] Trevor Fountain, Alexandra McCarthy, and Fangfang Peng. 2005. PCI express: An overview of PCI express, cabled PCI express and PXI express. In *Proceedings of the International Conference on Accelerator & Large Experimental Physics Control Systems (ICALEPCS'05)*.

[26] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdury, and Kang G. Shin. 2017. Efficient memory disaggregation with INFINISWAP. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI'17)*. 649–667.

[27] Anubhav Guleria, J. Lakshmi, and Chakri Padala. 2019. EMF: Disaggregated GPUs in datacenters for efficiency, modularity and flexibility. In *Proceedings of the International Conference on Cloud Computing in Emerging Markets (CCEM'19)*. 1–8. https://doi.org/10.1109/CCEM48484.2019.000-5

[28] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. 2017. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the International Systems and Storage Conference (SYSTOR'17)*. 1–9. https://doi.org/10.1145/3078468.3078483

[29] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balkrishnan. 2018. Performance characterization of NVMe-over-fabrics storage disaggregation. *ACM Trans. Stor.* 14, 4 (Dec. 2018), 1–18. https://doi.org/10.1145/3239563

[30] Steven Alexander Hicks, Michael Riegler, Konstantin Pogorelov, Kim V. Ånonsen, Thomas de Lange, Dag Johansen, Mattis Jeppsson, Kristin Ranheim Randel, Sigrun Eskeland, and Pål Halvorsen. 2018. Dissecting deep neural networks for better medical image classification and classification understanding. In *Proceedings of the International Symposium on Computer-Based Medical Systems (CBMS'18)*. 363–368. https://doi.org/10.1109/CBMS.2018.00070

[31] Rui Hou, Tao Jiang, Liuhang Zhang, Pengfei Qi, Jianbo Dong, Haibin Wang, Xiongli Gu, and Shujie Zhang. 2013. Cost effective data center servers. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'13)*. 179–187. https://doi.org/10.1109/HPCA.2013.6522317

[32] 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 the International Parallel and Distributed Processing Symposium (IPDPS'12)*. 774–785. https://doi.org/10.1109/IPDPS.2012.74

[33] Neo Jia and Kirti Wankhede. 2016. VFIO Mediated Devices. Retrieved from https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt.

[34] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, and Rajeev Thakur. 2004. High performance MPI-2 one-sided communication over InfiniBand. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGrid'04)*. 531–538. https://doi.org/10.1109/CCGrid.2004.1336648

[35] Linux kernel development community. [n.d.]. NTB Drivers. Retrieved from https://www.kernel.org/doc/html/latest/driver-api/ntb.html.

[36] Linux kernel development community. 2013. Linux Filesystems API. Retrieved from https://www.kernel.org/doc/htmldocs/filesystems/index.html.

[37] Linux kernel development community. 2013. VFIO—"Virtual Function I/O." Retrieved from https://www.kernel.org/doc/Documentation/vfio.txt.

[38] Linux kernel development community. 2019. Linux IOMMU Support. Retrieved from https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt.

[39] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*. 41–45.

[40] KaiGai Kohei. 2016. GpuScan + SSD-to-GPUDirect DMA. Retrieved from https://kaigai.hatenablog.com/entry/2016/09/08/003556.

[41] 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 the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'16)*. 10:1–10:6. https://doi.org/10.1145/2910642.2910650

[42] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. 2005. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'05)*. 1–10. https://doi.org/10.1109/CLUSTR.2005.347050

[43] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the the Annual International Symposium on Computer Architecture (ISCA'09)*. 267–278. https://doi.org/10.1145/1555754.1555789

[44] Seung-Ho Lim, Ki-Woong Park, and Kwang-Ho Cha. 2019. Developing an OpenSHMEM model over a switchless PCIe non-transparent bridge interface. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW'19)*. 593–602. https://doi.org/10.1109/IPDPSW.2019.00104

[45] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. 2013. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *Proceedings of the International Conference on Parallel Processing (ICPP'13)*. 641–650. https://doi.org/10.1109/ICPP.2013.78

[46] Evangelos P. Markatos and George Dramitinos. 1996. Implementation of a reliable remote memory pager. In *Proceedings of the USENIX Annual Technical Conference (ATC'96)*.

[47] Athanasios Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. 2019. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*. https://doi.org/10.14722/ndss.2019.23194

[48] Jonas Markussen, Lars Bjørlykke Kristiansen, Rune Johan Borgli, Håkon Kvale Stensland, Friedrich Seifert, Michael Riegler, Carsten Griwodz, and Pål Halvorsen. 2020. Flexible device compositions and dynamic resource sharing in PCIe interconnected clusters using Device lending. *Cluster Comput.* 23 (2020), 1211–1234. Issue 2. https://doi.org/10.1007/s10586-019-02988-0

[49] 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 *Proceedings of the International Conference on Parallel Processing Companion (ICPPComp'18)*. Article 48, 48:1–48:10. https://doi.org/10.1145/3229710.3229759

[50] Vijay Meduri. 2011. A Case for PCI Express as a High-Performance Cluster Interconnect. Retrieved from https://www.hpcwire.com/2011/01/24/a_case_for_pci_express_as_a_high-performance_cluster_interconnect/.

[51] Microsemi. 2019. *Multi-Host Sharing of NVMe Drives and GPUs Using PCIe Fabrics*. Technical Report. Microsemi. Retrieved from http://www.symmttm.com/document-portal/doc_download/1244483-multi-host-sharing-of-nvme-drives-and-gpus-using-pcie.

[52] Ben-Yehuda Muli, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, Jun Nakijima, and Elsie Wahlig. 2006. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the Linux Symposium*. 71–85.

[53] NVIDIA Corporation 2019. *GPUDirect RDMA Documentation*. NVIDIA Corporation. Retrieved from https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[54] NVIDIA Corporation 2020. *CUDA Toolkit Documentation v11.0.171*. NVIDIA Corporation. Retrieved from http://docs.nvidia.com/cuda/.

[55] NVM Express 2019. *NVM Express Base Specification*. NVM Express. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3d-2019.03.20-Ratified.pdf.

[56] NVM Express 2019. *NVM Express Over Fabrics*. NVM Express. Retrieved from https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf.

[57] Sinno Jialin Pan and Qiang Yang. 2010. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.* 22, 10 (Oct. 2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

[58] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: A NVMe storage virtualization solution with mediated pass-through. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*. 665–676.

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

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

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

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

[63] Konstantin Pogorelov, Olga Ostroukhova, Mattis Jeppsson, Håvard Espeland, Carsten Griwodz, Thomas de Lange, Dag Johansen, Michael Riegler, and Pål Halvorsen. 2018. Deep learning and hand-crafted feature based approaches for polyp detection in medical videos. In *Proceedings of the International Symposium on Computer-Based Medical Systems (CBMS'18)*. 381–386. https://doi.org/10.1109/CBMS.2018.00073

[64] Konstantin Pogorelov, Kristin Ranheim Randel, Carsten Griwodz, Sigrun Losada Eskeland, Thomas de Lange, Dag Johansen, Concetto Spampinato, Duc-Tien Dang-Nguyen, Mathias Lux, Peter Thelin Schmidt, Michael Riegler, and Pål Halvorsen. 2017. KVASIR: A multi-class image dataset for computer aided gastrointestinal disease detection. In *Proceedings of the ACM Multimedia Systems Conference (MMSys'17)*. 164–169. https://doi.org/10.1145/3083187.3083212

[65] Konstantin Pogorelov, Michael Riegler, Sigrun Eskeland, Thomas de Lange, Dag Johansen, Carsten Griwodz, Peter Thelin Schmidt, and Pål Halvorsen. 2017. Efficient disease detection in gastrointestinal videos–global features versus neural networks. *Multimedia Tools Appl.* 76, 21 (2017), 22493–22525. https://doi.org/10.1007/s11042-017-4989-y

[66] Konstantin Pogorelov, Michael Riegler, Jonas Markussen, Mathias Kux, Håkon Kvale Stensland, Thomas Lange, Carsten Griwodz, Pål Halvorsen, Dag Johansen, Peter Schmidt, and Sigrun Eskeland. 2016. Efficient processing of videos in a multi auditory environment using device lending of GPUs. In *Proceedings of the International Conference on Multimedia Systems (MMSys'16)*. 381–386. https://doi.org/10.1145/2910017.2910636

[67] Murali Ravindran. 2008. Extending cabled PCI express to connect devices with independent PCI domains. In *Proceedings of the IEEE Systems Conference (SysCon'08)*. 1–7. https://doi.org/10.1109/SYSTEMS.2008.4519048

[68] Carlos Reaño, Federico Silla, and José Duato. 2017. Enhancing the rCUDA remote GPU virtualization framework: From a prototype to a production solution. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID'17)*. 695–698. https://doi.org/10.1109/CCGRID.2017.42

[69] Jack Regula. 2004. *Using Non-Transparent Bridging in PCI Express Systems*. Whitepaper. PLX Technology/Broadcom. Retrieved from https://www.digikey.no/no/pdf/b/broadcom/using-non-transparent-bridging-pci.

[70] Davide Rosetti. 2014. Benchmarking GPUDirect RDMA on Modern Server Platforms. Retrieved from https://developer.nvidia.com/blog/benchmarking-gpudirect-rdma-on-modern-server-platforms/.

[71] Andy Rudoff. 2017. Persistent memory programming. *USENIX ;login:* 42, 2 (2017), 34–40. Retrieved from https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf.

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

[73] Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. Retrieved from https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/.

[74] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI'18).* 69–87.

[75] Cheol Shim, Kwang-Ho Cha, and Min Choi. 2018. Design and implementation of initial OpenSHMEM on PCIe NTB based cloud computing. *Cluster Comput.* 22 (Feb. 2018), 1815–1826. https://doi.org/10.1007/s10586-018-1707-0

[76] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. Retrieved from https://arXiv:1409.1556.

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

[78] Jun Suzuki, Yoichi Hidaka, Hunichi Higuchi, Masaki Kan, and Takashi Yoshikawa. 2016. Disaggregation and sharing of I/O devices in cloud data centers. *IEEE Trans. Comput.* 65 (Dec. 2016), 3013–3026. Issue 10. https://doi.org/10.1109/TC.2015.2513759

[79] 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 the IEEE Symposium on High Performance Interconnects (HOTI'10).* 25–31. https://doi.org/10.1109/HOTI.2010.21

[80] Amir Taherkordi, Feroz Zahid, Yiannis Verginadis, and Geir Horn. 2018. Future cloud system designs: Challenges and research directions. *IEEE Access* 6 (2018). https://doi.org/10.1109/ACCESS.2018.2883149

[81] Mellanox Technologies. [n.d.]. ConnectX-5 EN Single/Dual-Port Adapter Supporting 100Gb/s Ethernet. Retrieved from https://www.mellanox.com/products/ethernet-adapters/connectx-5-en.

[82] PLX Technologies. 2005. *Multi-Host System and Intelligent I/O Design with PCI Express.* Whitepaper. PLX Technology/Broadcom. Retrieved from https://docs.broadcom.com/docs-and-downloads/pdf/technical/expresslane/NTB_Brief_April-05.pdf.

[83] Adam Thompson and Chris J. Newburn. 2019. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. Retrieved from https://developer.nvidia.com/blog/gpudirect-storage/.

[84] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. 2011. A case for RDMA in clouds. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys'11).* 17:1–17:5. https://doi.org/10.1145/2103799.2103820

[85] Shin-Yeh Tsai and Yiying Zhang. 2019. A double-edged sword: Security threats and opportunities in one-sided network communication. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud'19).*

[86] Cheng-Chun Tu. 2014. *Memory-Based Rack Area Networking.* Ph.D. Dissertation. Stony Brook University.

[87] Cheng-Chun Tu and Tzi-cker Chiueh. 2018. Seamless fail-over for PCIe switched networks. In *Proceedings of the International Systems and Storage Conference (SYSTOR'18).* 101–111. https://doi.org/10.1145/3211890.3211895

[88] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. 2013. Secure I/O device sharing among virtual machines on multiple hosts. *ACM SIGARCH Comput. Architect. News* 41, 3 (2013), 108–119. https://doi.org/10.1145/2508148.2485932

[89] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. 2014. Marlin: A memory-based rack area network. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'14).* 125–136. https://doi.org/10.1145/2658260.2658262

[90] Akshay Venkatesh, Khaled Hamidouche, Sreeram Potluri, Davide Rosettig, Ching-Hsiang Chu, and Dhabaleswar K. Panda. 2017. MPI-GDS: High performance MPI designs with GPUDirect-aSync for CPU-GPU control flow decoupling. In *Proceedings of the International Conference on Parallel Processing (ICPP'17).* 151–160. https://doi.org/10.1109/ICPP.2017.24

[91] Akshay Venkatesh, Hari Subramoni, Khaled 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 the International Conference on High Performance Computing (HiPC'14).* 1–10. https://doi.org/10.1109/HiPC.2014.7116875

[92] Heymian Wong. 2011. *PCI Express Multi-Root Switch Reconfiguration During System Operation.* Master's thesis. Massachusetts Institute of Technology.

[93] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'20).* 169–182.

[94] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A development kit to build high performance storage applications. In *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom'17).* 154–161. https://doi.org/10.1109/CloudCom.2017.14

[95] Xiangliang Yu. 2016. NTB: Add support for AMD PCI-Express Non-Transparent Bridge. Retrieved from https://lwn.net/Articles/672752/.

Received July 2020; revised February 2021; accepted April 2021