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, overprovisioning 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

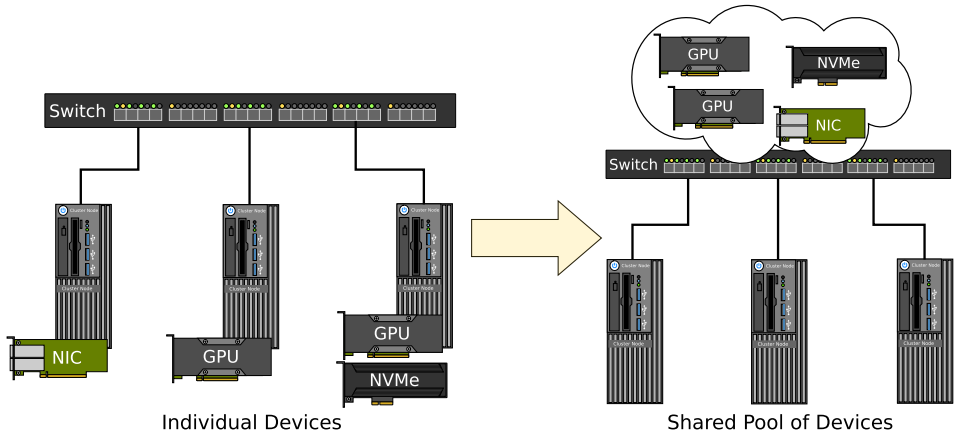


Fig. 1. SmartIO allows the internal devices of hosts in the network to be shared with other hosts connected to the same fabric. Nodes in a PCIe-networked cluster can contribute their internal devices to a shared device pool, and borrow resources from that pool when needed.

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.

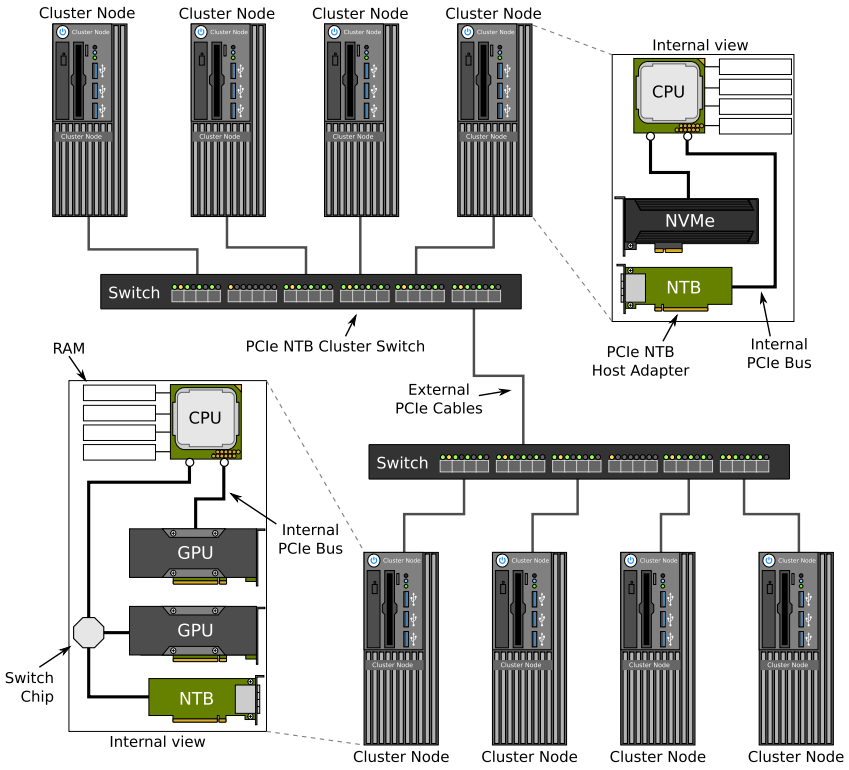


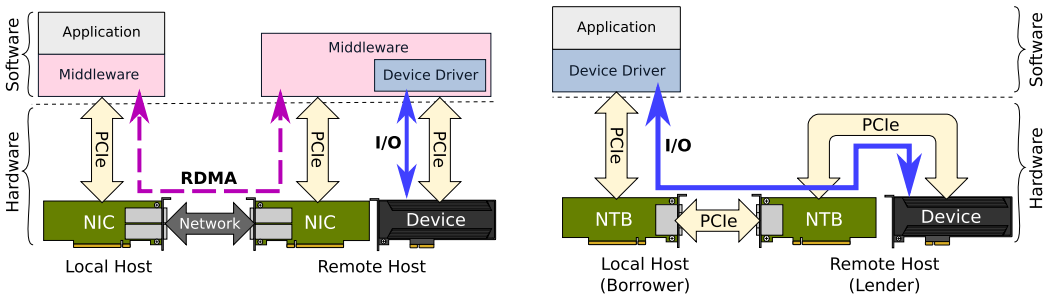
Fig. 2. We can create a heterogeneous PCIe cluster by interconnecting nodes (hosts) with external PCIe links using adapter cards capable of non-transparent bridging (NTB). In such clusters, the CPUs as well as the internal devices of each node are all attached to the same PCIe network fabric.

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.

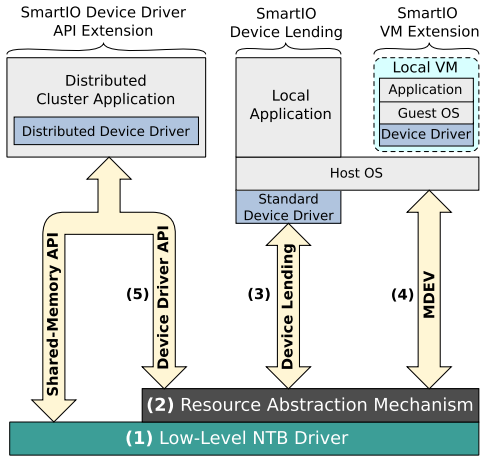


Fig. 4. SmartIO provides different interfaces that facilitate access to a remote resource. These interfaces present an abstraction layer to application software and device drivers, providing a logical decoupling of devices and which physical hosts they are installed in.

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.

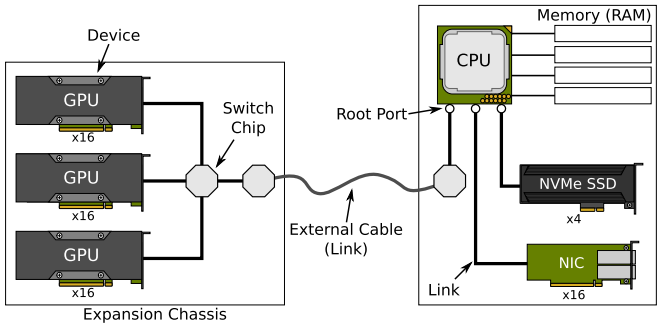


Fig. 5. Example of a PCIe topology using an external link to connect an expansion chassis to a computer system. The devices in the expansion chassis are part of the same PCIe tree as the internal devices, because all downstream links (including the external cable) are *transparent*.

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).

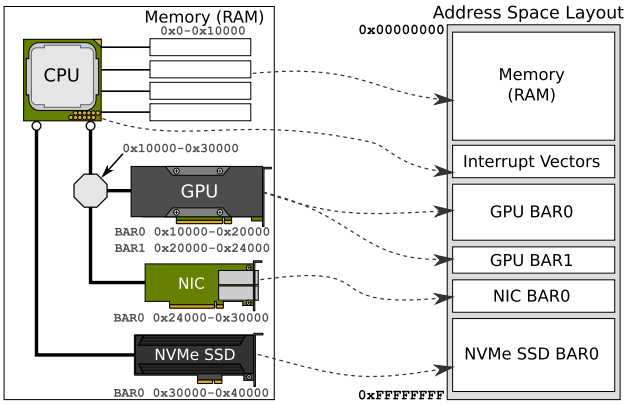


Fig. 6. Device memory regions (BARs) are mapped into the same address space as system memory.

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.

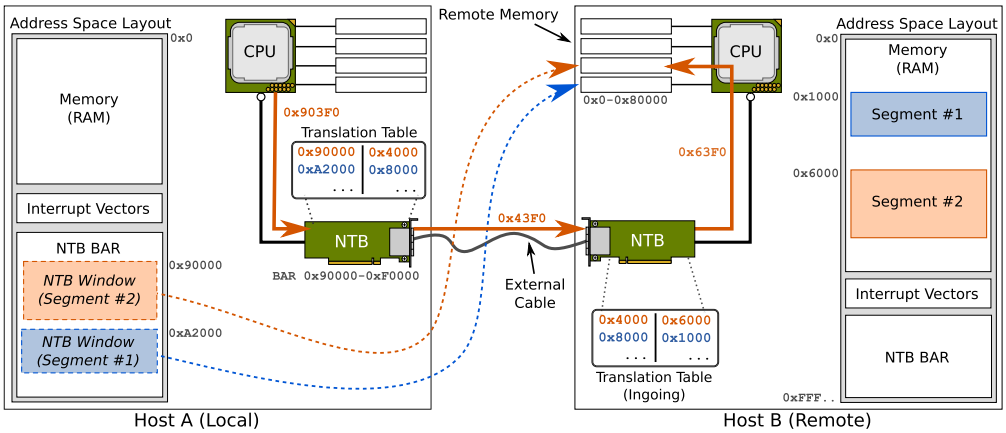


Fig. 7. Example of two independent PCIe root complexes connected together using an NTB. The link between the two hosts is *non-transparent*, and the NTB translates addresses between the two domains. Host A has mapped parts, or segments, of Host B's memory through its local NTB, providing Host A with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

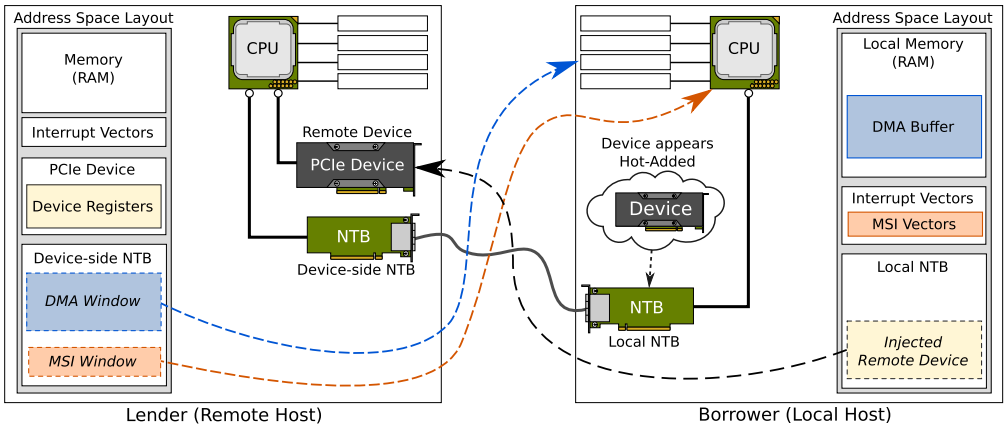


Fig. 8. Device Lending: Using NTBs, it is possible to map the memory regions of a remote device so a local CPU can access device registers. The remote system can in turn reverse-map local resources for the device, making DMA and MSI possible. Device Lending injects a hot-added "shadow device" into the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

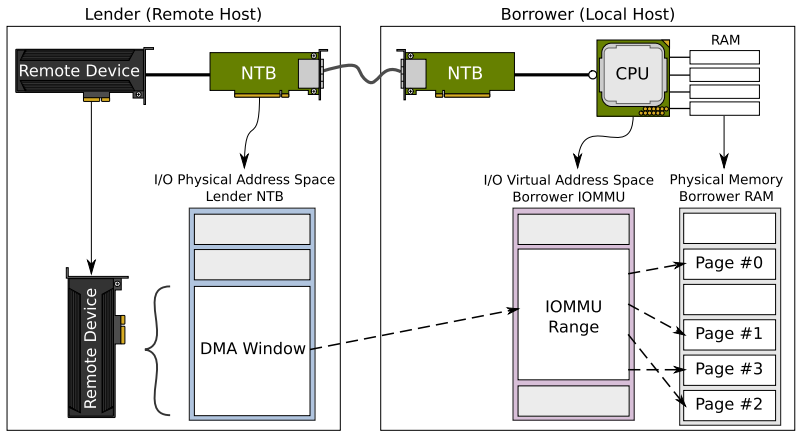


Fig. 9. DMA window: We use the local IOMMU in order create a single continuous memory range. This allows us to conserve NTB resources by setting up a single mapping through the device-side NTB in order for the remote device to reach local RAM. Adding and removing memory pages from the local IOMMU group 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

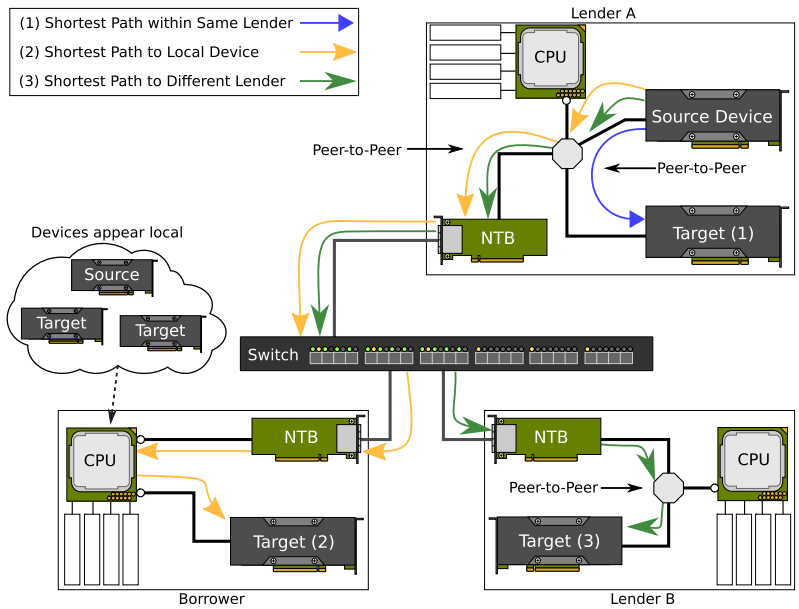


Fig. 10. Shortest path routing: By resolving addresses of device memory regions and preparing mappings for them in advance, we can route device-to-device using the shortest path when a device driver initiates a DMA transfer. Our solution covers all three scenarios: (1) when both devices are in the same 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.

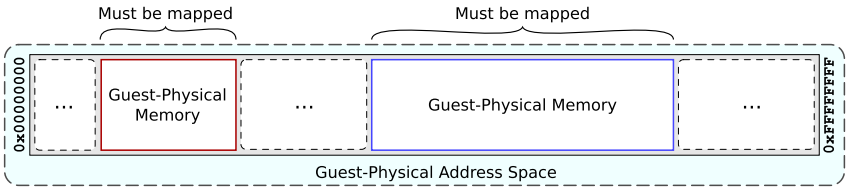


Fig. 11. Mapping VM memory for a device: The VM's address space may be much larger than the actual memory used by the guest. Only guest-physical memory needs to be mapped for a device.

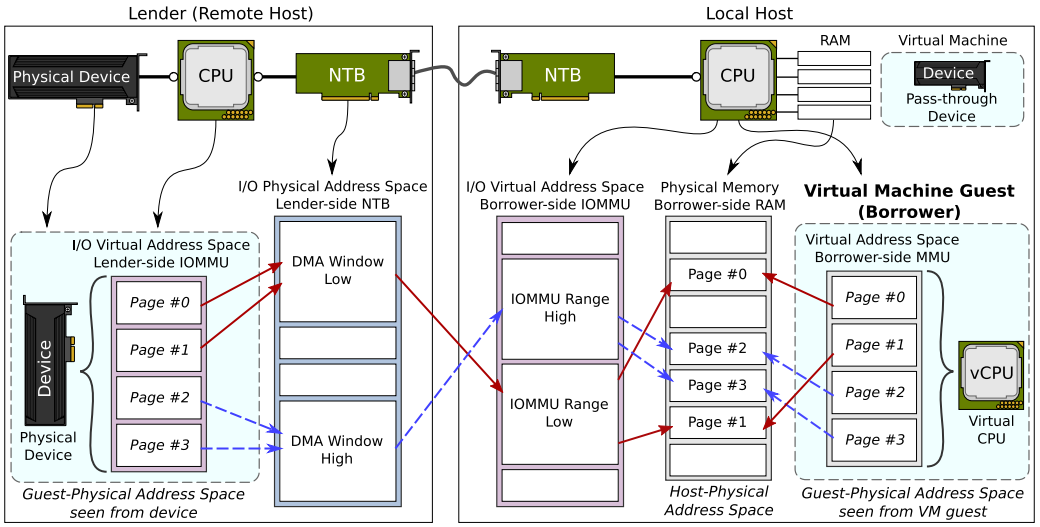


Fig. 12. Pass-through of a remote device: By using IOMMUs on both sides of the NTB, it is possible to map a remote device into a local VM guest's address space. The borrower-side IOMMU provides continuous memory ranges that can be mapped over the NTB, while the lender-side IOMMU is used to map the virtual 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.

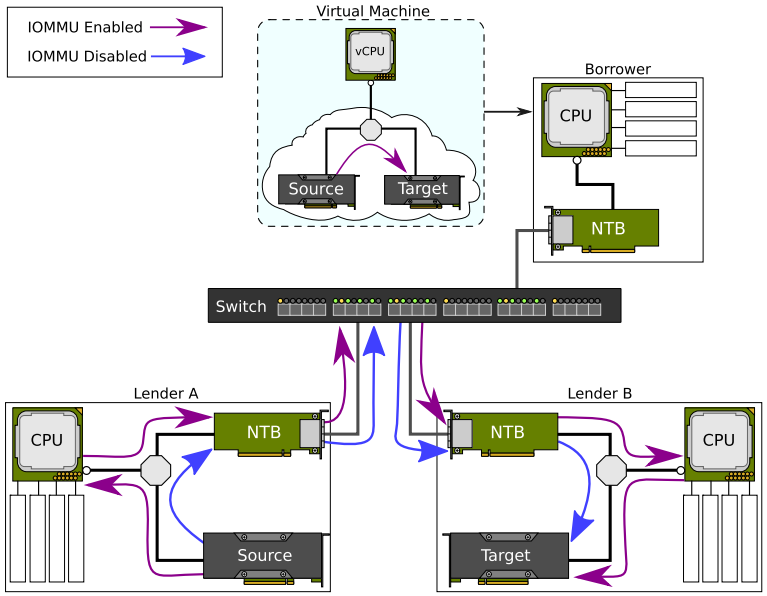


Fig. 13. Since IOMMUs introduce a virtual address space for devices, peer-to-peer transfers must be routed through the root in order for the IOMMU to resolve virtual addresses to physical addresses. As a consequence, shortest path routing is disrupted.

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.

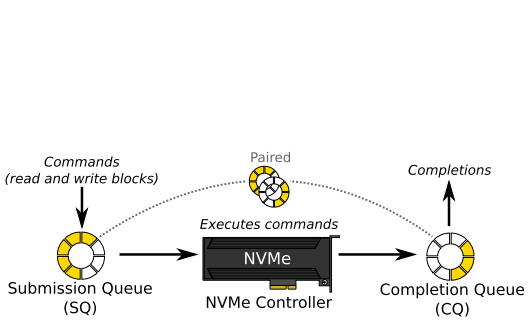
6.1 Device Driver API

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.

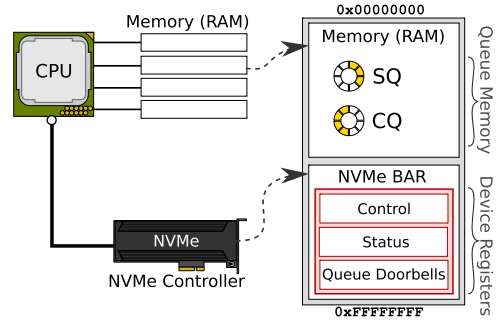
As mentioned in Section 3.3, a userspace application may map "segments" of a remote system's memory into its own virtual address space using SISCI. Moreover, as we explained in Section 4.3, we can set up mappings to such shared memory segments for a *device* as well ("DMA windows"). Devices may use DMA 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:

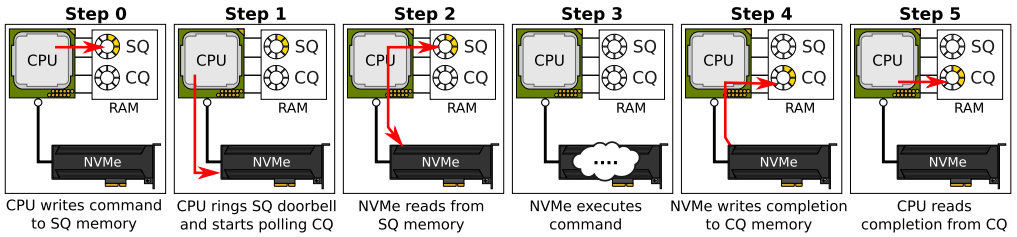
- API functions for letting application processes borrow and return devices. Borrowing a device can either be exclusive, allowing only one borrower at the time, or non-exclusive, allowing several borrowers simultaneously. It possible for a single application process to first take an exclusive reference, 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.



(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.

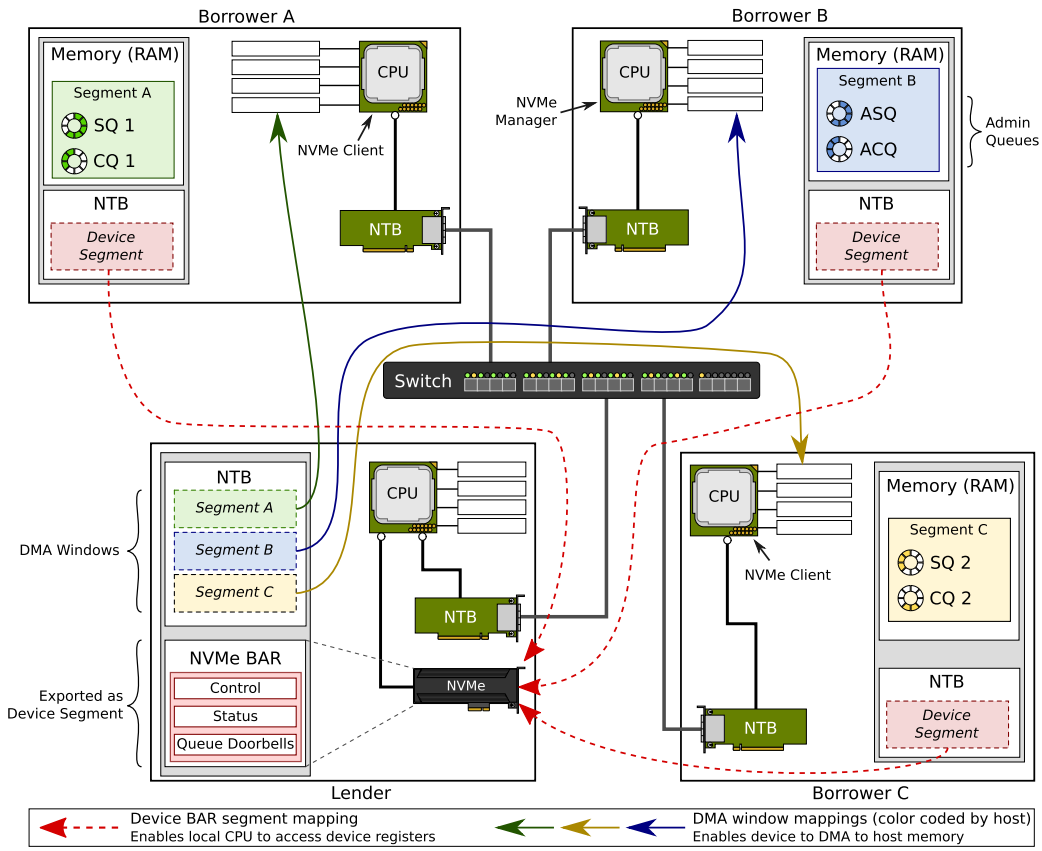


Fig. 15. Simultaneous sharing: The NVMe device can access queues residing in memory segments on different hosts by mapping the segments for the device (DMA windows). Likewise, the borrowers must in turn map the doorbell registers for their respective queues to notify the device about queue events. Each 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

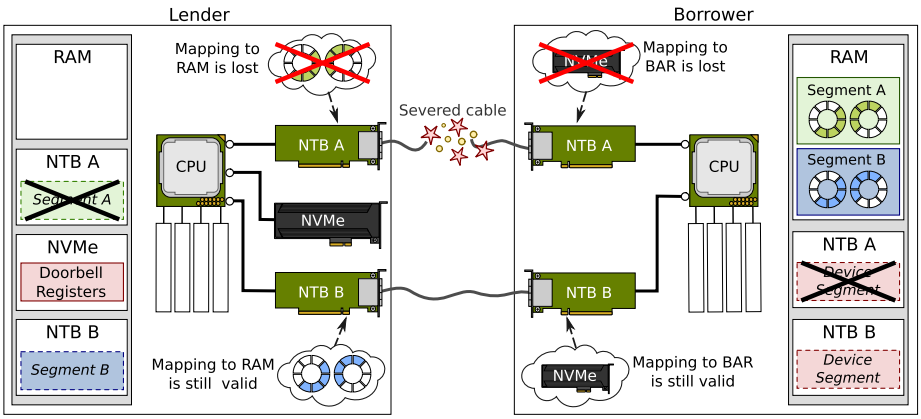


Fig. 16. Multipath failover: We can configure multiple NVMe queue pairs and mapping their memory for the device through different NTB adapters. Similarly, we can also map doorbell registers through separate adapters for the borrower. By having different paths for each I/O queue pair, we can continue 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.

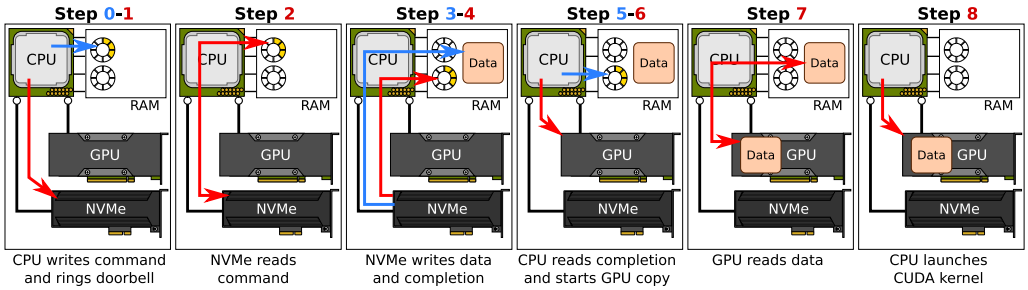
The link may become active again with invalid mappings. In the case of I/O queue pairs, this is inconsequential as the NVMe standard supports deleting and creating I/O queues during operation; we can simply delete the old queues, set up new DMA windows and create new queues. However, special care must 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

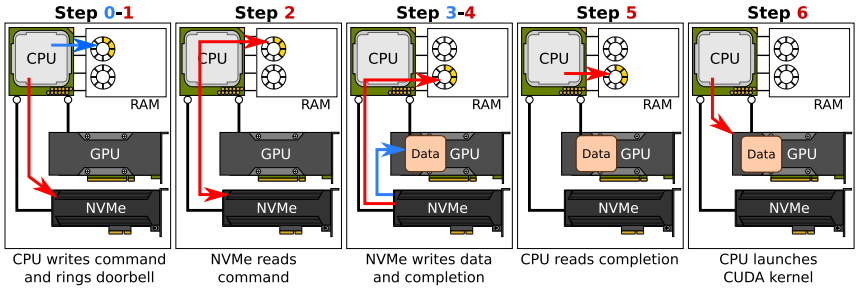
Many GPU-accelerated applications require fast access to storage. For example, the datasets in big data and machine learning tasks can be hundreds of terabytes. As datasets' size for typical GPU workloads is only increasing, GPU applications become bounded by transfers between storage and GPU. To overcome 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.

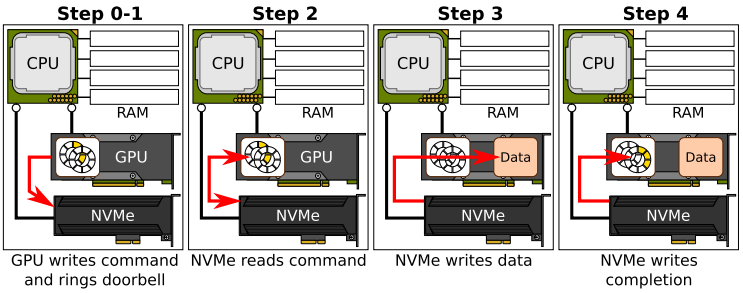


Fig. 18. Avoiding CPU synchronization: By hosting I/O queues in GPU memory and mapping doorbell registers for the GPU, a CUDA kernel running on the GPU can operate the NVMe without involving the CPU.

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

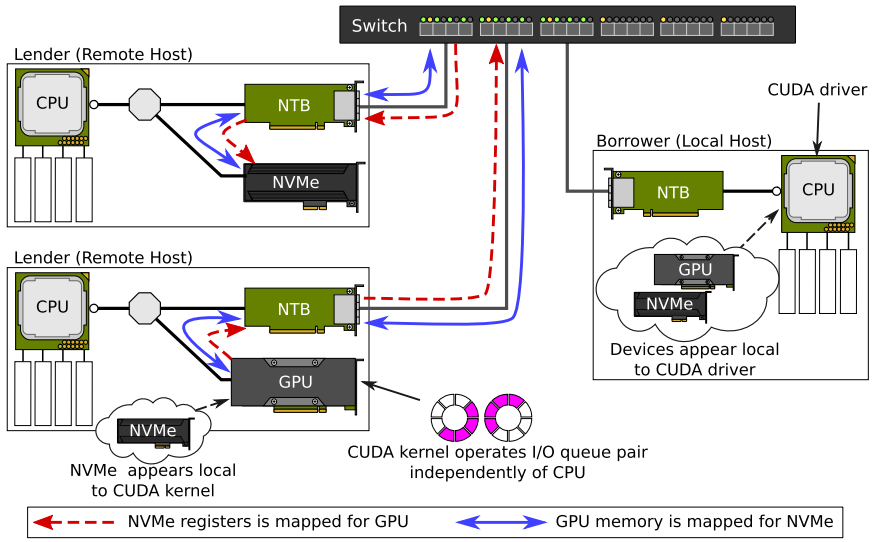


Fig. 19. By combining the SmartIO API extension and Device Lending, our NVMe driver supports direct access to a remote NVMe from a borrowed GPU. To the CUDA driver running on the local system, both the NVMe device and GPU appear local. Our SmartIO system injects necessary peer-to-peer mappings transparently. 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

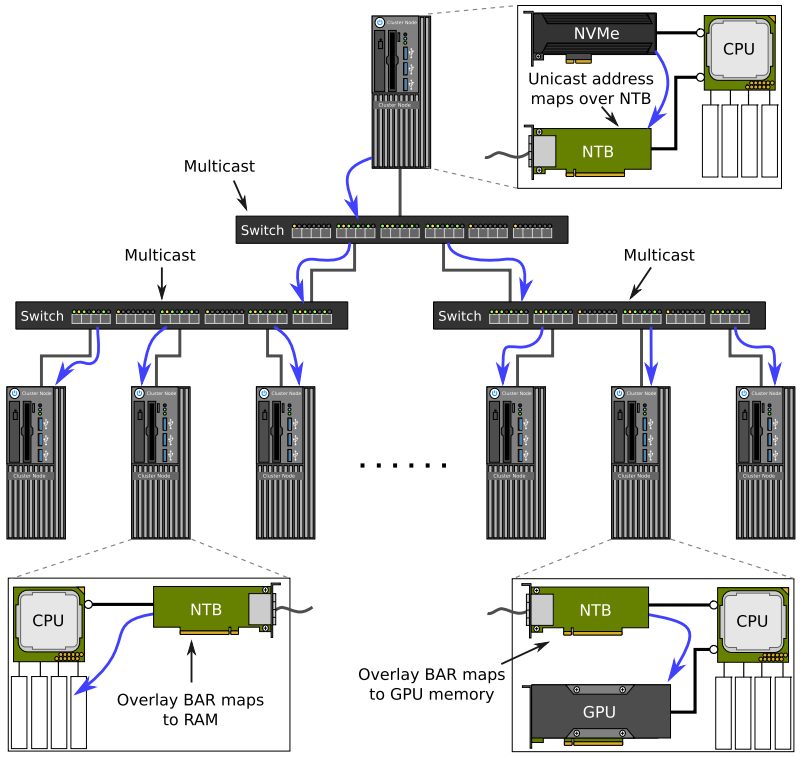


Fig. 20. Multicast support makes it possible for a single DMA operation to be replicated to multiple destinations. It is possible to map multicast destination to system memory and device memory alike.

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.

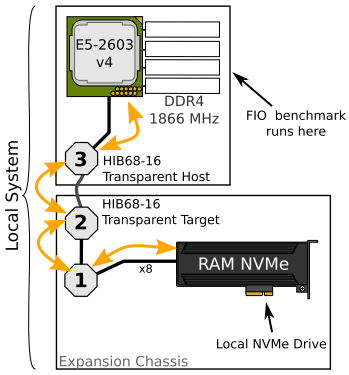
7.1.1 Latency Tests. To prepare a DMA transfer, memory must be mapped for a device. This involves locking pages in memory so they are not swapped out and resolving their I/O addresses. For reading from block device, i.e., a storage device, the Linux block-layer pin the pages used by a memory buffer and 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.

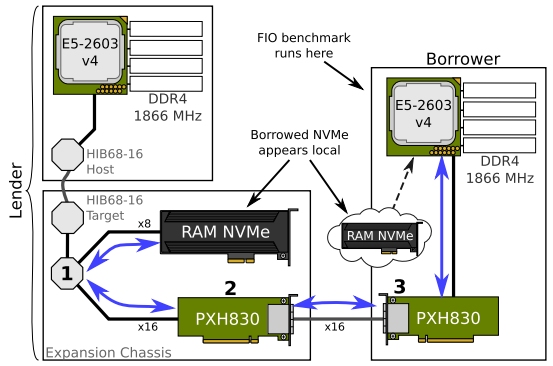
To measure any performance impact of our mapping routine, we have used the ***Flexible I/O tester* (FIO)** [9]. FIO is a widely used userspace application for benchmarking the performance of storage devices, such as NVMe devices. By configuring FIO to perform reads and using the *sync* engine, FIO opens 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:

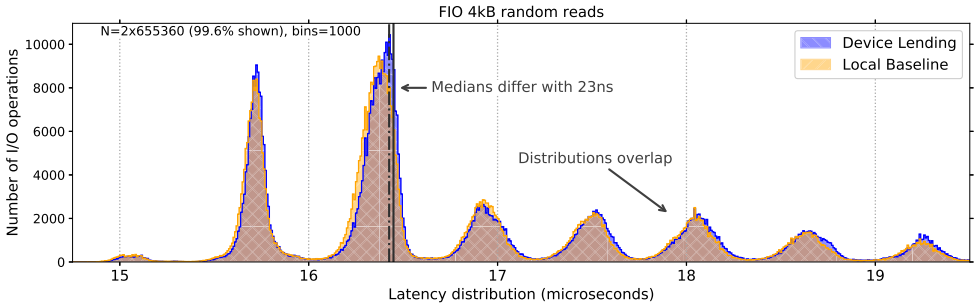
- **Local Baseline**, shown in Figure 21(a): An expansion chassis connected to a local host running CentOS 7, using the 3.10 version of the kernel and the built-in NVMe driver. We are running FIO version 3.7 as available from the CentOS 7 software repositories.
  The expansion chassis is connected to 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.

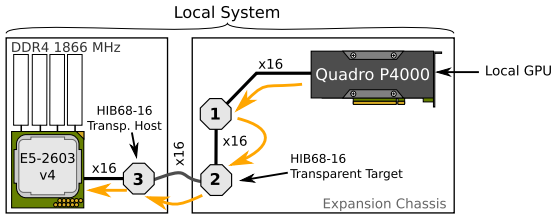
Figure 21(c) shows the latency distribution of read operations for both using a local NVMe device (Local Baseline) and when accessing a remote NVMe device using Device Lending. Although the purpose of the test is simply to compare Device Lending to local access, it is interesting to note that the distributions 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].

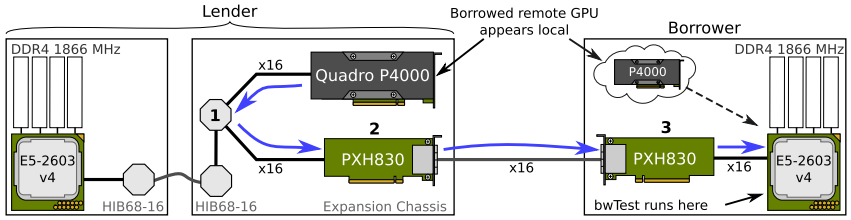
By performing large DMA transfers, we saturate the PCIe links with DMA traffic and also stress system memory. This allows us to investigate if there is any performance difference between using a local device or a borrowed, remote device for high-throughput workloads. Any overhead caused by our IOMMU 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

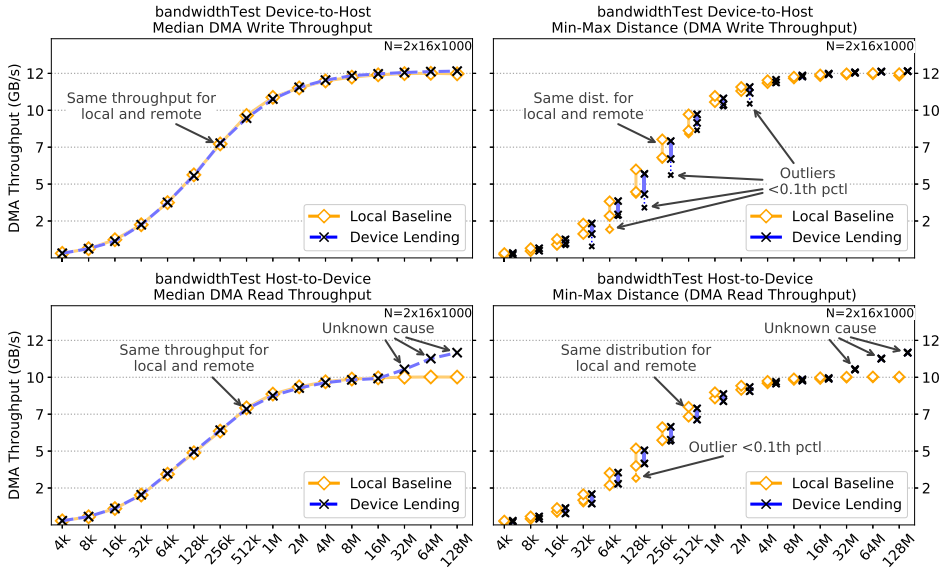
[9]Since an NVMe read operation involves a register write, several DMA transactions and interrupts, comparing similar hardware topologies would also reveal any latency overhead in the address translation mechanism of the NTBs as well.



(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.

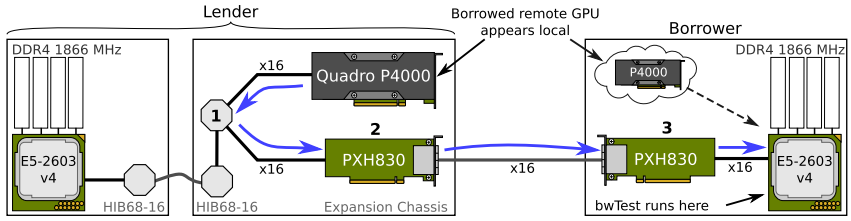
- **Device Lending**, shown in Figure 22(b): Two hosts connected back-to-back using Dolphin PXH830 NTB adapter cards, one host is borrowing the Quadro P4000 GPU. The IOMMU on the lender host is disabled, to enable DMA transfers to be routed shortest path over the NTB in the expansion chassis, making 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.

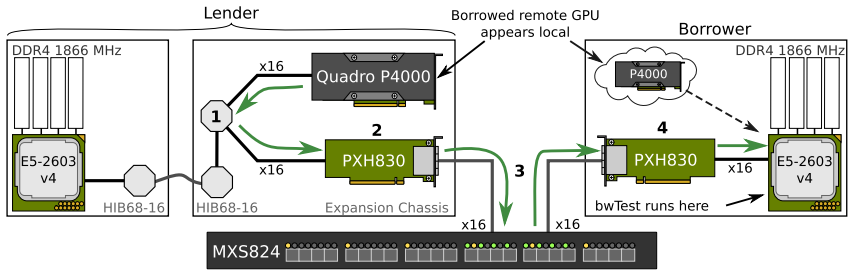
Figure 22(c) depicts the results of our test, with DMA writes in the top row and DMA reads in the bottom row. The different transfer sizes are plotted along the X-axis. The left column depicts the median of 1,000 runs. To show that even the distribution of measurements are similar for local and remote, 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.

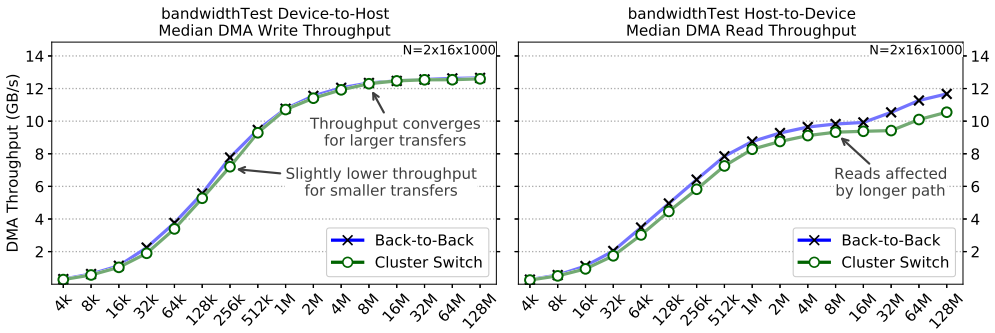
[10]The maximum payload size for a device is configured by the system. While it can be configured individually for each device, it is usually configured to be the same for all devices in the PCIe tree due to several practical reasons, and is most commonly set to 128, 256, or 512 bytes.



(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.

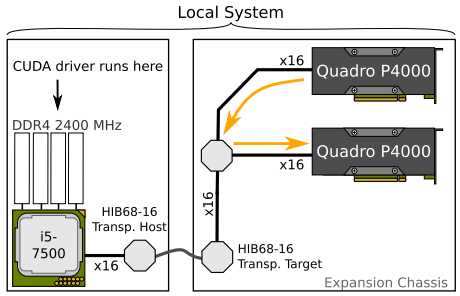
DMA reads suffer noticeably from the increased distance. Unlike writes, which are posted transactions, the number of read requests simultaneously being held open is limited. Moreover, PCIe allows a completer to respond with less data at the time than is actually requested. For example, a read requesting 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:

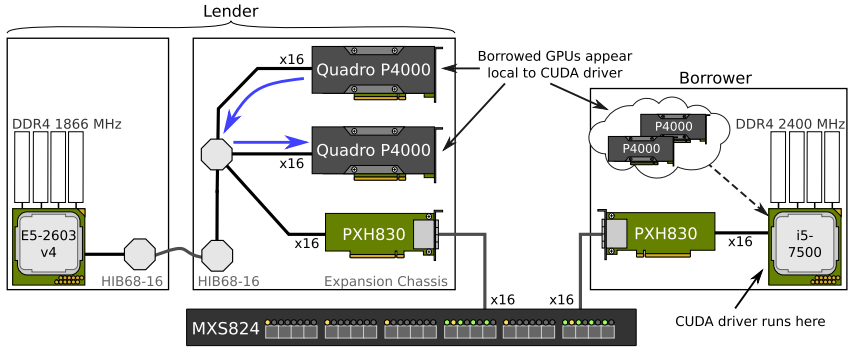
- **Local Baseline**, shown in Figure 24(a): A local system using two local GPUs in an expansion chassis. We have disabled the IOMMU on the local CPU, to enable shortest path routing within the expansion chassis.
- **Device Lending**, shown in Figure 24(b): Two hosts connected together using Dolphin 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).

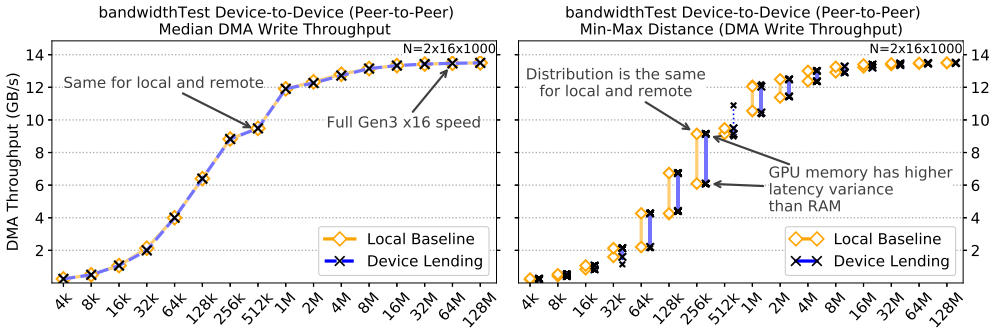
Using the same topologies as depicted in Figure 24, we have also measured the latency of DMA writes between the two GPUs. We developed a small CUDA program to measure peer-to-peer latency, as depicted in Figure 25(a). One GPU is tasked with increasing a counter, writing it to the other GPU's memory and 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.

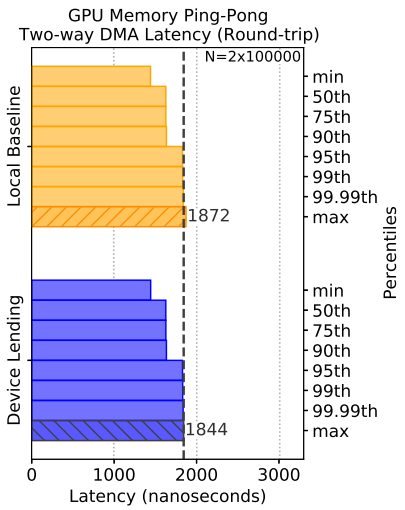
Fig. 24. Peer-to-peer throughput: We demonstrate that our Device Lending implementation supports shortest path routing by comparing peer-to-peer DMA performance. The IOMMU on the borrowing system does not affect traffic between borrowed devices.

```
__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.

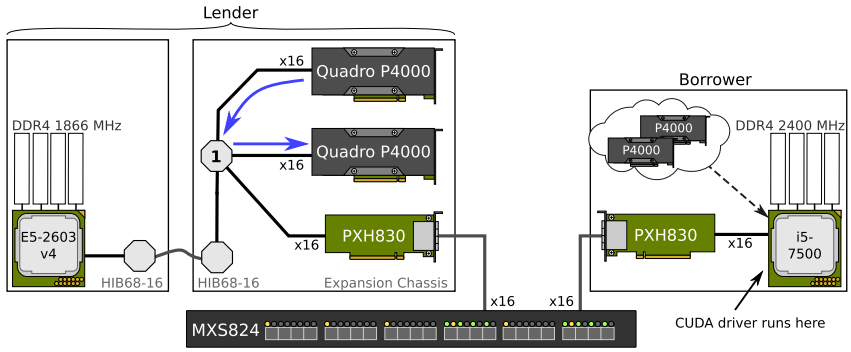
Fig. 25. Peer-to-peer latency: by implementing a ping-pong program in CUDA, we can measure the latency of DMA writes between two GPUs. One GPU writes a 4-byte message to the other GPU's memory, before waiting for an acknowledgement and recording the time before and after (ping). The other GPU waits for 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.

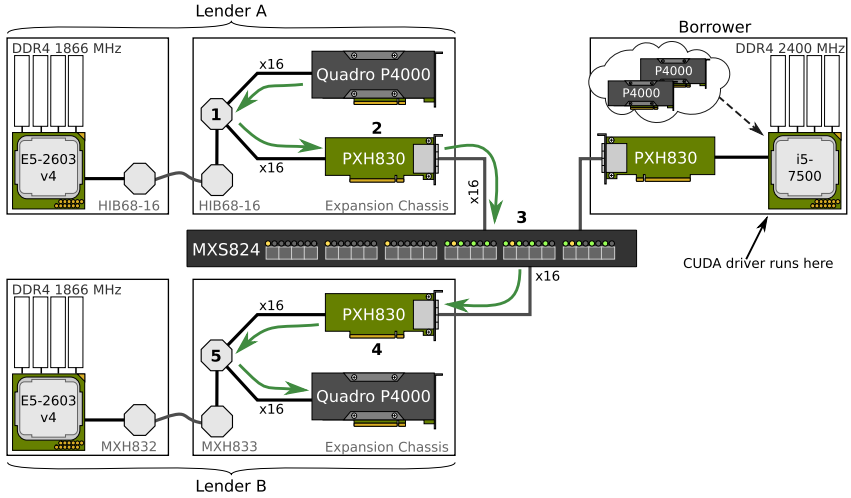
7.1.5 Peer-to-peer: Multiple Lenders. As described in Section 4.4, our Device Lending implementation also supports shortest-path routing between devices even when they reside in *different* lender systems. By composing a PCIe infrastructure consisting of devices spread out over multiple hosts in the 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:

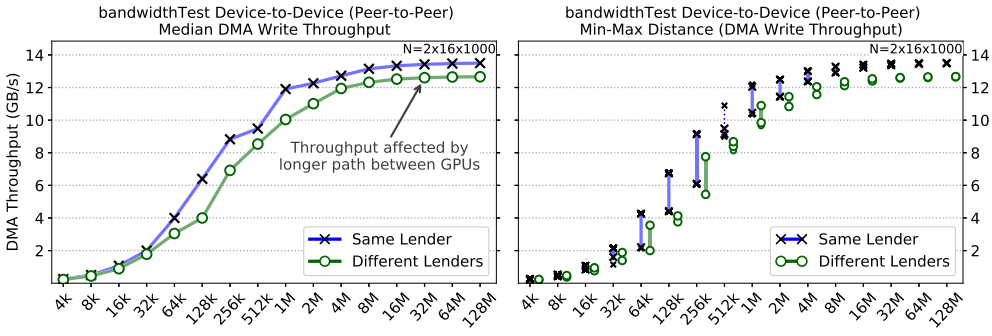
- **Same Lender**, shown in Figure 26(a): Using two GPUs from the same lender. As we established in the previous section, this scenario is similar to a local system using local devices.
- **Different Lenders**, shown in Figure 26(b): Using two GPUs from different lenders. DMA transactions have to traverse 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.

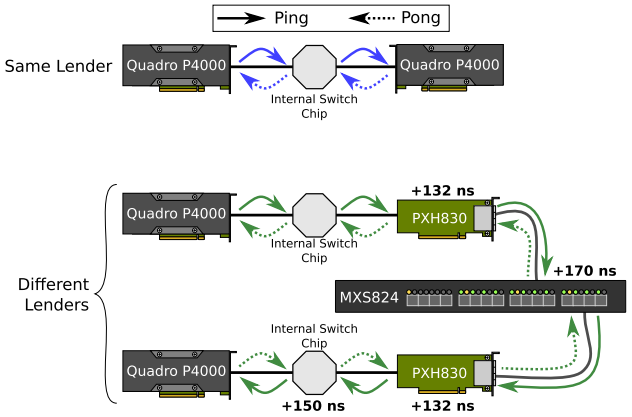


(b) **Different Lenders**: using two GPUs from different systems. The data path of one GPU writing to the memory of the other is illustrated. Note the increased number of hops.

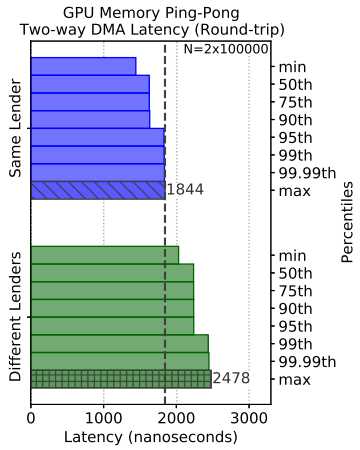


(c) The median throughout of DMA writes from one GPU to another GPU (left), and the throughput distribution of all writes shown as min–max distance (right). We can see that moving the second GPU to a different system, thus increasing the distance, affects performance.

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.

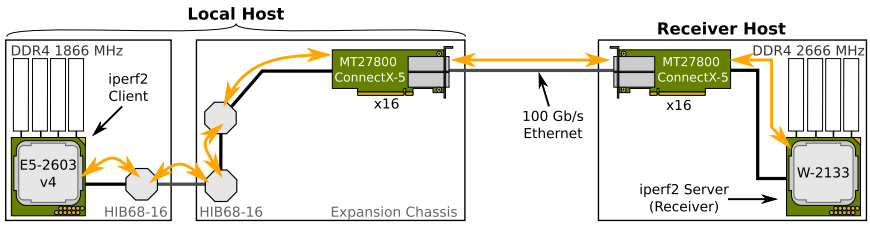
Figure 28 depicts the configuration used in these tests, where the client connects to the server running on the receiver host:

- **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.

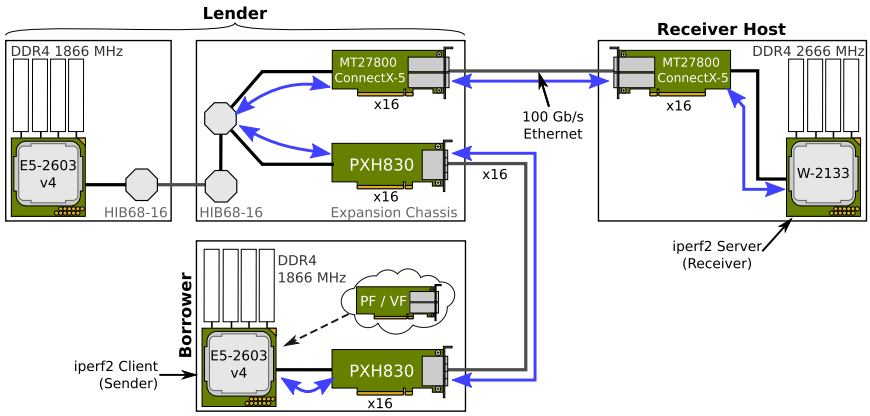
All hosts run Ubuntu 18.04 with the 4.15 version of the Linux kernel, using the in-kernel Mellanox Ethernet driver. To compare apples to apples, we have disabled the IOMMU on both lender and borrower, as well as on the receiver host. In all cases, the iperf2 client runs for a duration of five minutes, 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

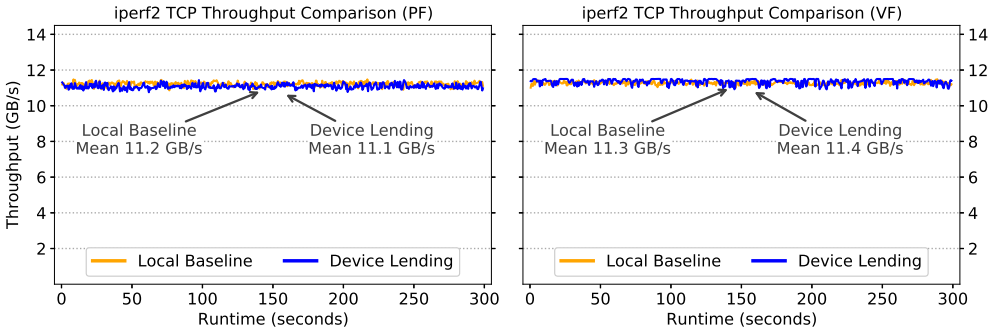
[11]The behavior of iperf2 is perhaps counter-intuitive. In most client/server applications, the client will typically request data from the server rather than the server acting as a receiver. We have used the same terminology as the program uses.



(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.

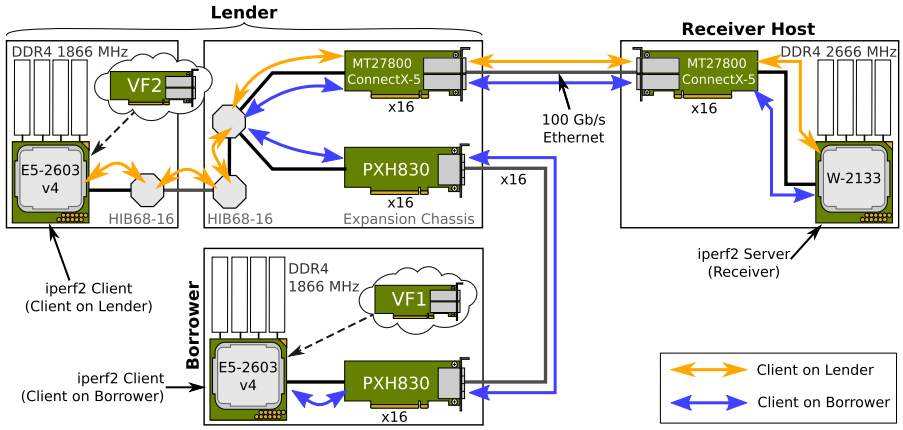


(c) TCP throughput at 1/2 second intervals for both the physical and virtual functions. Observe that there is no significant performance difference between the Local Baseline and Device Lending. Additionally, hardware virtualization (VF) does not impact performance compared to the physical device function (PF).

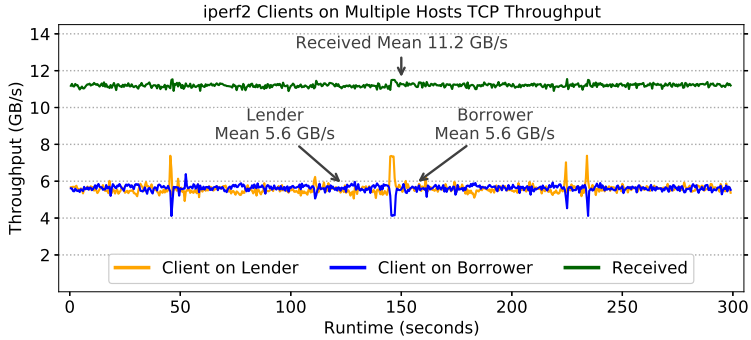
Fig. 28. TCP throughput comparison: We compare the achieved throughput for a client/server application.

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.

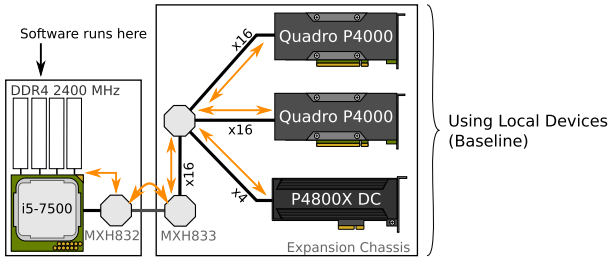
Our workload is a typical convolutional neural network training using the Python machine learning framework Keras [1]. Keras is a high level framework that wraps different lower level machine learning frameworks. In our case, Keras uses Tensorflow [2] as its back-end. Keras also allows multiple GPUs 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 gastroaintestinal tract called Kvasir [30, 63, 64] to perform disease classification [65].

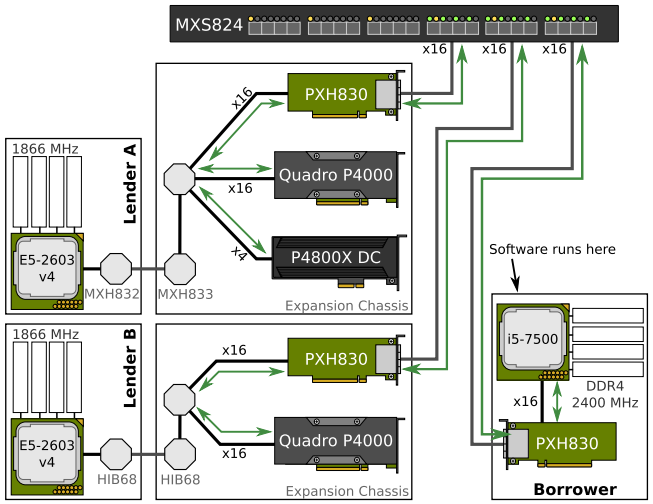
We measure the runtime of 12 epochs of the model training on two Nvidia P4000 GPUs as well as loading images from storage and writing the results back using an Intel Optane P4800X NVMe device. While 12 epochs may not give the statistical significance needed for reliable machine learning results, we are 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).

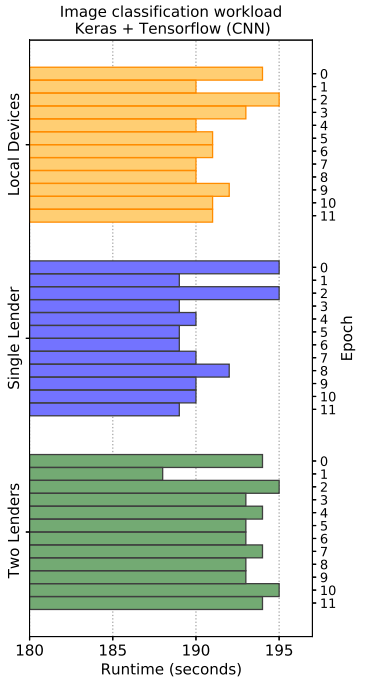
Our machine learning workload proves it is possible to use Device Lending for realistic workloads in a PCIe cluster, dynamically creating configurations of both local and remote devices and accessing them without any performance penalty beyond what is expected for longer PCIe paths. We argue that this 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:

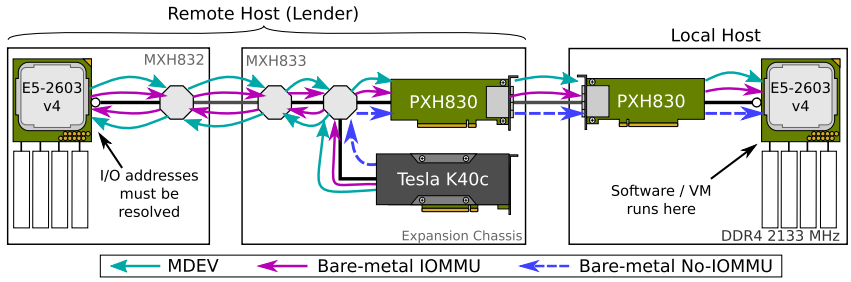
(1) Reads suffer particularly from the increased distance, as addresses are routed through the lender's CPU *twice* per transaction; the first time in order for the IOMMU to translate the addresses of the read requests, and the second time for completions with the requested data.
(2) By using a PCIe 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.

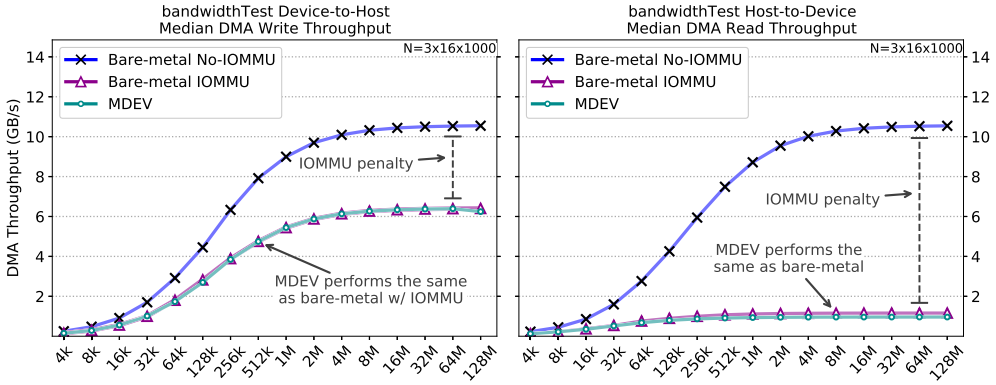
7.3.2 Pass-through Comparison. We have also repeated the same peer-to-peer benchmarks described in Section 7.1 using VMs. By using the peer-to-peer benchmarks to measure throughput and latency between two GPUs, we are able to compare our MDEV extension using remote devices to "normal" VFIO pass-through 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.



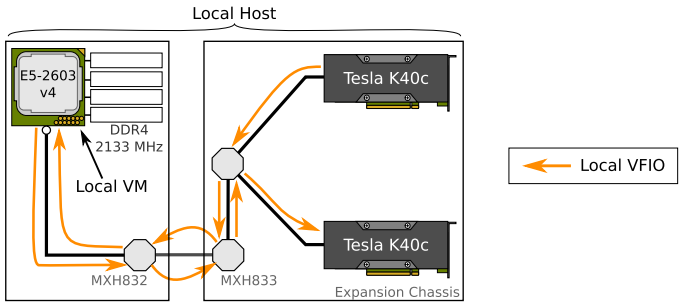
(b) Median throughput of DMA writes (left) and DMA reads (right). While enabling the lender's IOMMU significantly decreases performance, note that our MDEV implementation achieves the same performance as the bare-metal comparison (with the lender-side IOMMU enabled). This indicates that our implementation 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.

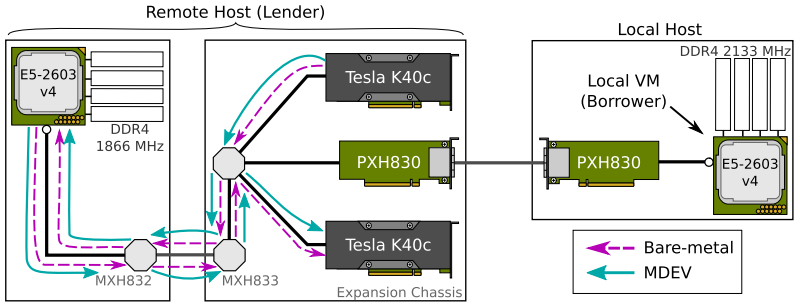
Both VM instances were configured with 4 GB memory, and we enabled 2 MB huge pages on the host. We also set the CPU affinity to be the same as the local adapter in both the bare-metal and MDEV scenarios.

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.



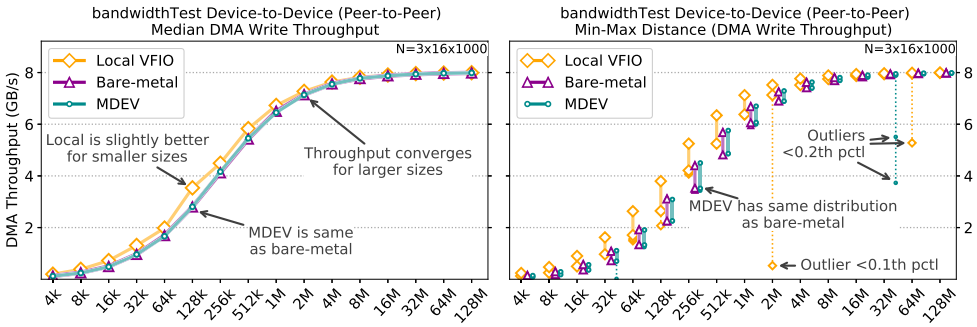
(b) **Bare-metal** and **MDEV**: two GPUs being used by a remote bare-metal machine with our Device Lending implementation and by a remote VM with our KVM+MDEV pass-through implementation. The direction of DMA writes from one GPU to the other is shown. Note that the data path is similar to the local 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.

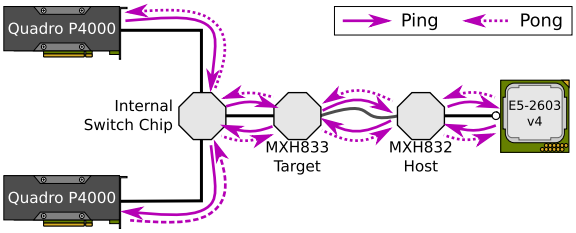
Figure 33(c) shows the distribution ping–pong latency measurements using the CUDA program we described in Section 7.1.4, where two GPUs writes a counter back and forth to each other's memory. The maximum measurement for MDEV appears to be an outlier, so we have annotated the 99.99th percentiles instead. 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

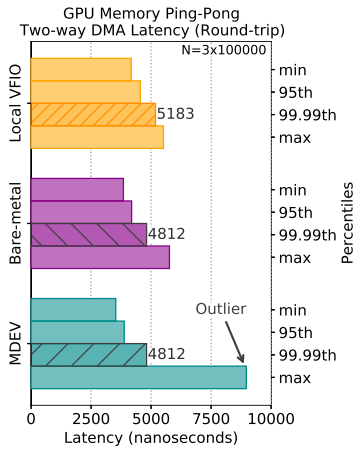
Our Device Lending and MDEV extension make it possible for a local device driver to operate a remote device in a manner that is fully transparent to both device and driver. This is possible as we prepare memory mappings in advance and inject addresses that map over the respective NTBs. However, as the 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.

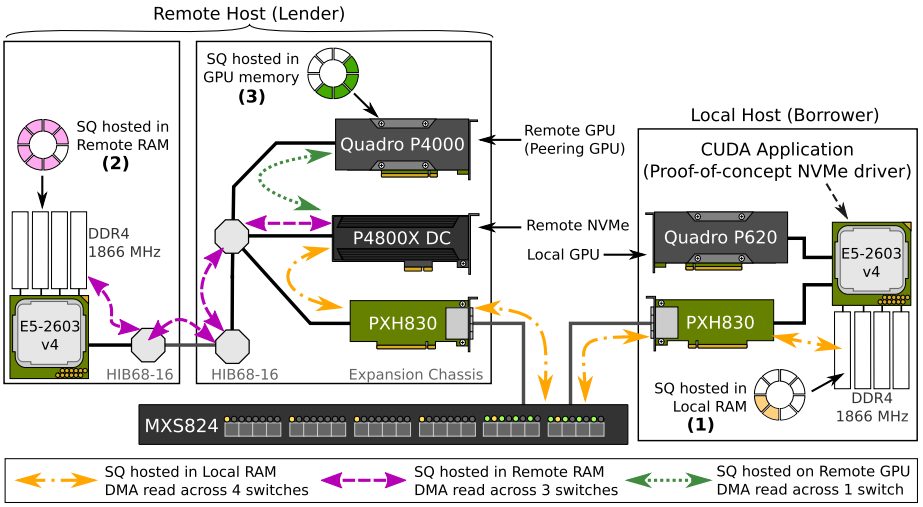
7.4.1 Optimizing Data Access Patterns. We outlined our userspace NVMe driver implementation using the SmartIO SISCI API extension in Section 6. Not only are we able to assign individual queues to different nodes, but we are also able use GPUDirect-capable GPUs to host queues in GPU memory as explained 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.

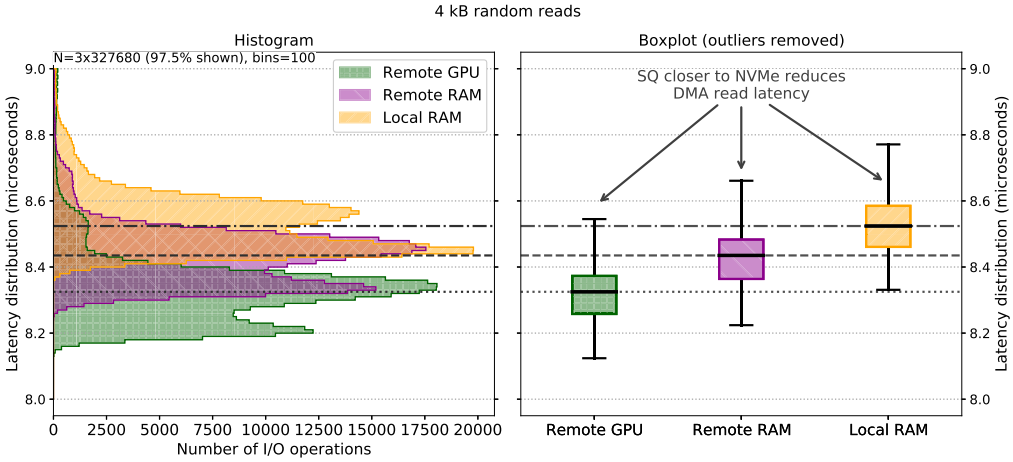
To evaluate the performance benefit of this strategy, we have designed the following experiment: a local CPU runs our proof-of-concept userspace NVMe driver (implemented as a CUDA application). It uses a local Nvidia Quadro P620 GPU and a remote Intel Optane P4800X DC NVMe device. The local GPU is managed 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.



(a) Our SmartIO NVMe driver makes it is 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,

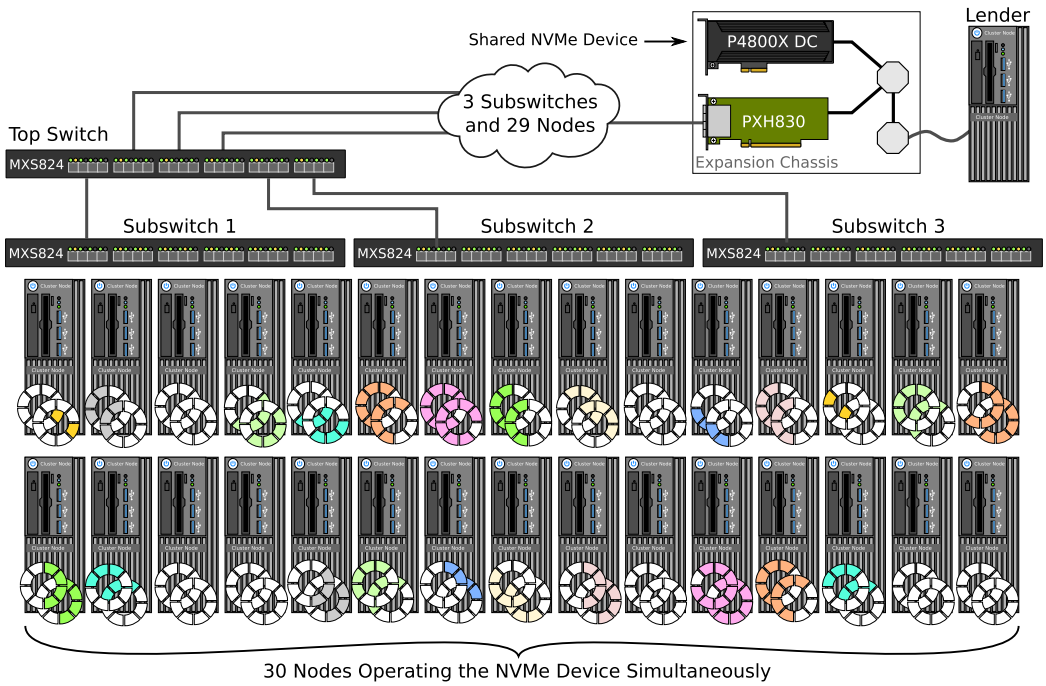


Fig. 35. By distributing an SQ and a CQ to 30 cluster nodes, we demonstrate that it is possible to concurrently share a single-function NVMe device in a larger cluster.

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 × 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.

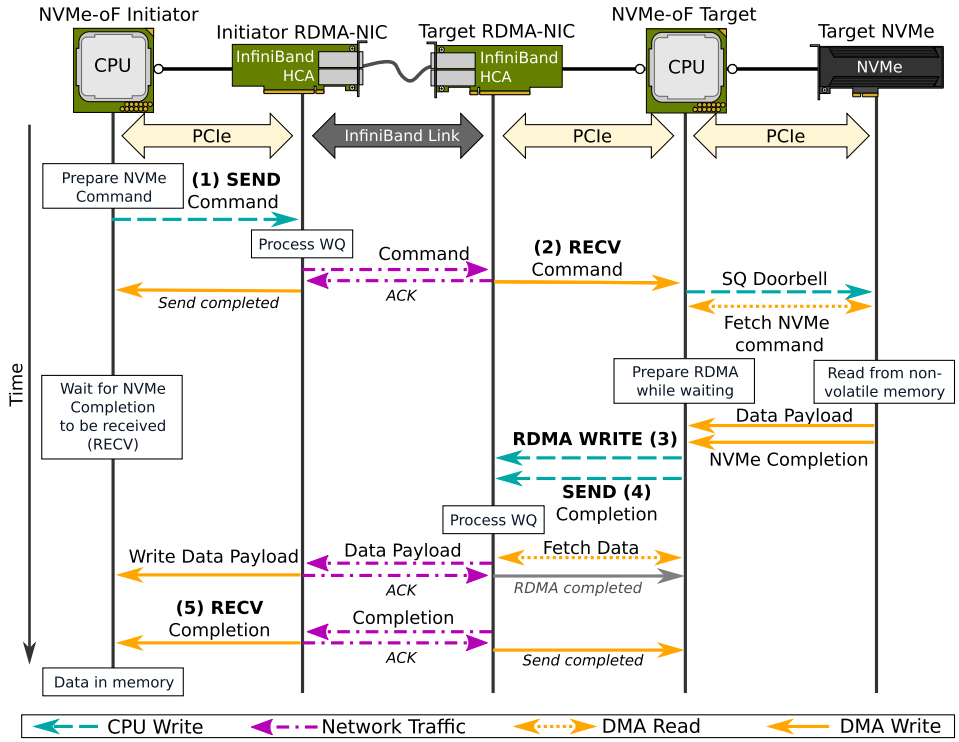


Fig. 36. Flow chart of an I/O read operation for NVMe-oF using InfiniBand RDMA. While the target-side CPU is required to initiate NVMe operations and start the RDMA write transfer, neither commands, completions, nor data is moved by the CPU. As InfiniBand queues and NVMe queues are bound to each other, 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.

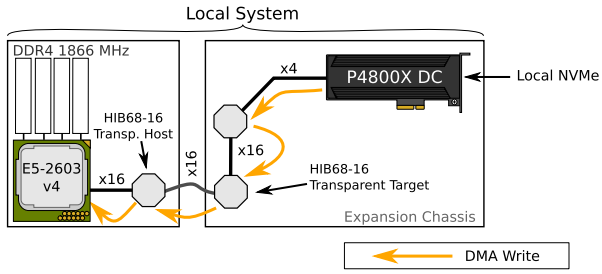
We have designed an experiment to compare the **Storage Performance Development Kit (SPDK)** [94] to our SmartIO NVMe driver implementation. SPDK is a storage application framework that implements support for a wide variety of storage devices, including NVMe devices. Similarly to our SmartIO NVMe driver, 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:

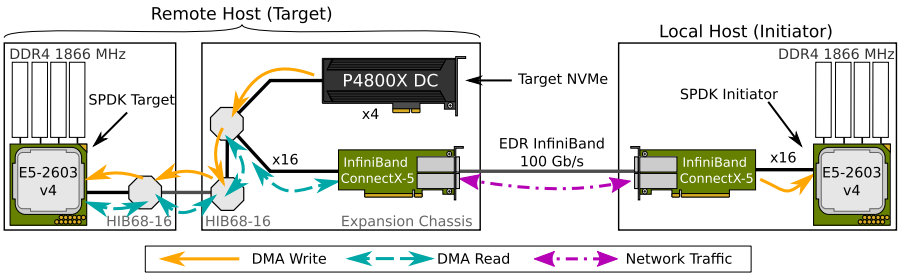
- **Local SPDK**, shown in Figure 37(a): The standard SPDK NVMe driver operating a local Intel Optane P4800X DC. The NVMe is installed in an expansion chassis, and connected upstream using the HIB68-16 transparent adapters. This scenario serves as our *local baseline* comparison for SPDK.
- **SPDK NVMe-oF**, 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.

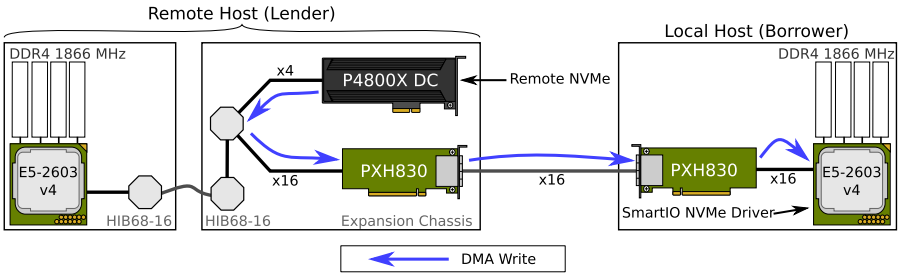
To measure read latency, we used FIO version 3.13 [9] to perform 327,680 reads, each read page-sized chunk (4 kB) with an offset generated by a pseudo-random number generator. Figure 38



(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

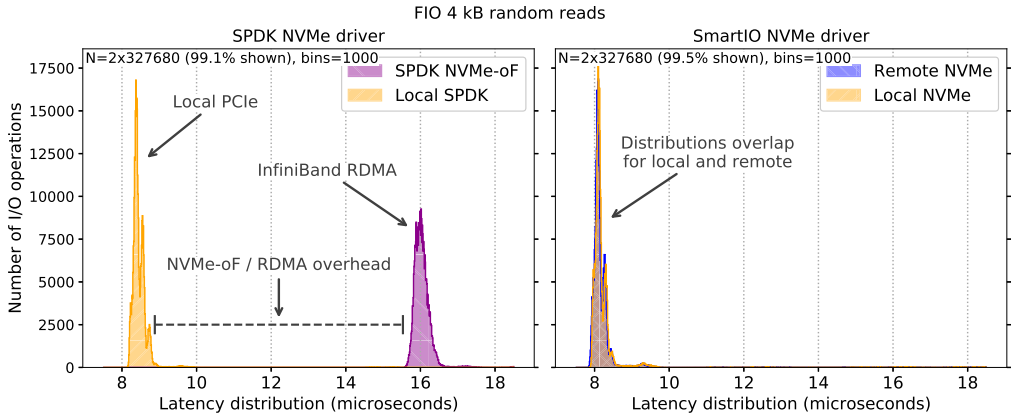


Fig. 38. Distribution of I/O command completion latencies for Local SPDK and SPDK NVMe-oF (left) and using our proof-of-concept SmartIO NVMe driver (right). By avoiding the device-side CPU in the critical path, as well as being able to use DMA directly, our NVMe driver achieves the same performance for 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

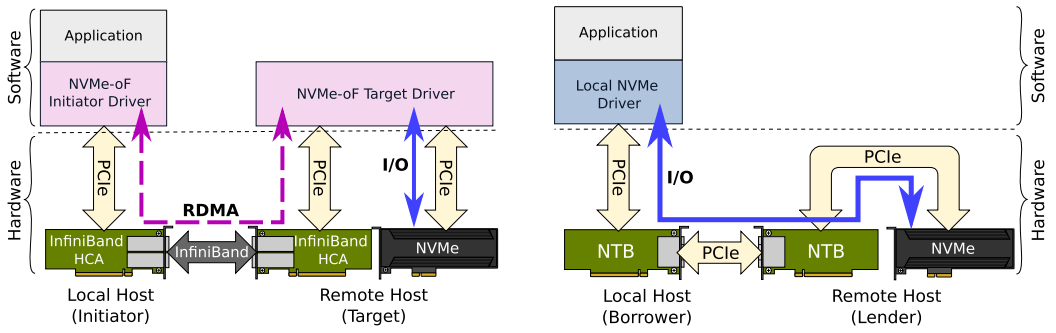
devices and assists the other hosts in setting up NTB mappings. However, instead of leveraging the NTB to map device BARs for the local host and mapping memory for the device, their solution appears to be based on transferring data from memory to memory, and then involving the local device driver on 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