NVMe over PCIe Fabrics using Device Lending

Jonas Markussen
jonassm@dolphinics.com

Lars Bjørlykke Kristiansen
larsk@dolphinics.com

Hugo Kohmann
hugo@dolphinics.com

August 2, 2019

Outline

The emerging standard for accessing remote NVMe drives today is NVMe over Fabrics (NVMeoF). By relying on remote direct memory access (RDMA), NVMeoF is able to facilitate efficient access to remote storage devices with very little overhead. However, encapsulating I/O commands, forwarding them using a network transport protocol and receiving them on the other end has an unavoidable latency cost compared to accessing a local device.

For computers that are connected in a Dolphin PCIe cluster, it is possible to access remote NVMe drives directly using a mechanism called Device Lending. Device Lending decouples devices from the hosts they physically reside in, allowing them to be dynamically assigned to any host in the cluster. To the local system, the NVMe drive appears local, allowing application software as well as native device drivers to use the drive without being aware that it is remote.

In this paper, we show how we can eliminate the inherent latency penalty of RDMA protocols by using Dolphin's SmartIO technology and accessing a NVMe drive directly. We compare our results to state of the art NVMeoF over Infiniband (IB).

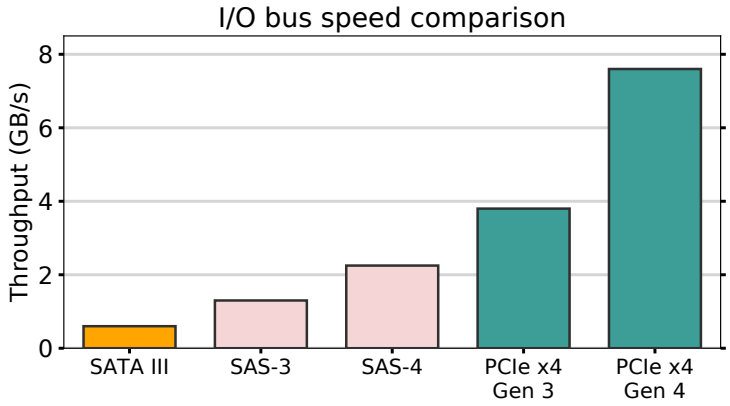


Figure 1: Throughput of PCIe compared to other traditional I/O buses. Consumer NVMe drives today are able to saturate a PCIe x4 bus for sequential reads.

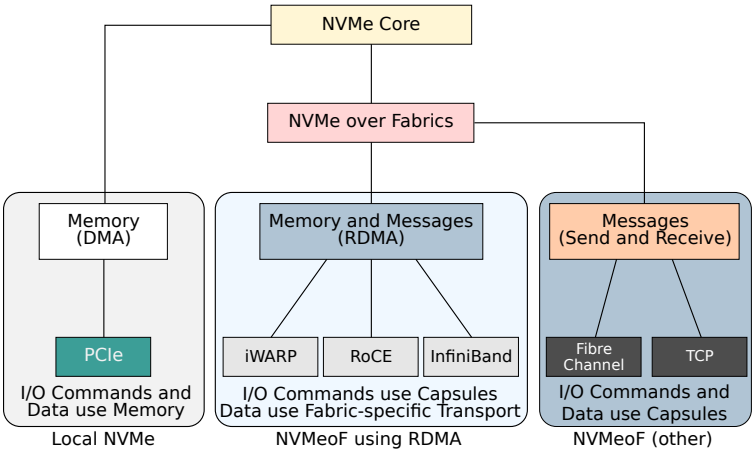
1 Motivation

Non-volatile storage media, such as flash memory solid-state drives (SSDs), have significantly lower latency and support higher I/O operations per second (IOPS) than traditional mechanical hard disks. Traditional I/O buses have become inadequate, and only the PCI Express (PCIe) bus is able to support the increased demands to throughput and latency. Inside a local computer, Non-Volatile Memory Express (NVMe) is the industry standard for interfacing with such PCIe-attached SSDs [1, 2]. By supporting parallel operations and relying on the memory addressing capabilities of PCIe, NVMe optimizes I/O command submission and data transfer paths.

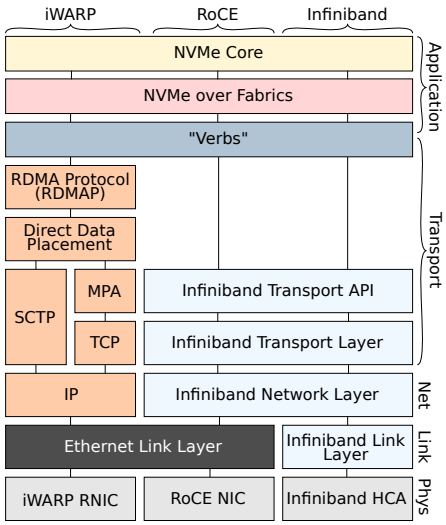
The throughput of the most commonly used I/O buses are shown in Figure 1. With some NVMe drives today being able to saturate a PCIe x4 Gen3 bus, the bottleneck of networked storage is no longer the storage drives, but has instead moved to the network. By building on top of the NVMe architecture and providing additional definitions for message-based NVMe operations, NVMe over Fabrics (NVMeoF) is the emerging protocol for accessing storage over a network [3]. By using NVMe semantics and providing support for remote direct memory access (RDMA), NVMeoF aims to achieve very little additional latency compared to local NVMe. However, some overhead is inevitable; encapsulating I/O commands in RDMA messages and translating to a network protocol introduce additional latency compared to native memory accesses.

For machines that are connected in a Dolphin PCIe cluster, it is possible to access remote devices using a mechanism we call Device Lending [4, 5]. Device Lending allows devices to be temporarily assigned to other machines in the cluster. To the system "borrowing" a remote device, the device appears locally installed. By "borrowing" a remote NVMe drive, application software, operating system and even device drivers may use the drive directly, without being aware that it is actually remote.

In this paper, we present the Device Lending mechanism as an alternative to NVMeoF using RDMA. Using a standard Linux kernel with default NVMe and NVMeoF drivers, we show how this approach compares to state-of-the-art NVMeoF using RDMA. Particularly, we focus on latency, and show that Device Lending achieves performance that is comparable to that of a local NVMe drive. We also briefly explain how we have extended the SISCI API with Device Lending semantics, and how we have used this to create a software library for enabling software-defined zero-copy data access to NVMe drives in the cluster.



(a) Taxonomy of NVMe transport fabrics. Fabrics that support RDMA can provide very high performance compared to message-based transport.



(b) Different RDMA architectures shown in the layered OSI model.

Figure 2: Architectural illustration of NVMeoF using RDMA.

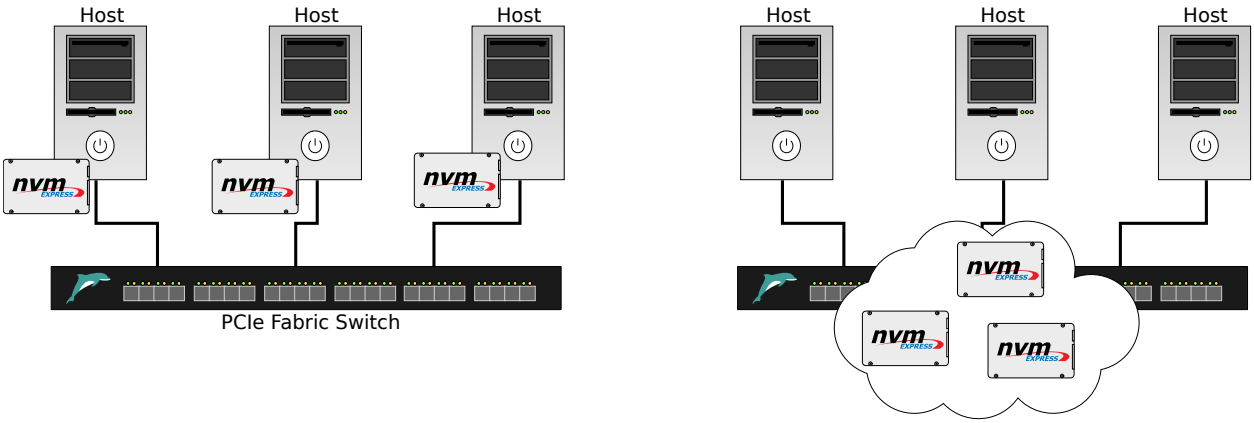
2 Taxonomy of NVMe over fabrics

In a local system, I/O command queues are mapped to memory within the host and accessible to a drive over the PCIe bus. By using direct memory access (DMA), a drive may fetch commands, read or write data depending on the command, and post completions directly to memory. This design reflects modern computer architectures, allowing NVMe drives to support parallelism through multiple queues as well as having very low latency by eliminating the need for hardware polling and synchronization. However, most networking technologies rely on message-based communication, with send and receive semantics, without the capability of sharing memory between the endpoints.

The NVMeoF specification defines an architecture for accessing remote NVMe drives and other block-level storage devices over a network ("fabric"). I/O commands and completions are encapsulated into "capsules", suitable for being transmitted over any message-passing communication channel while remaining independent of any specific networking fabric technology. Figure 2a shows an example of three supported fabric transports: NVMeoF over Fibre Channel, NVMeoF over TCP, and NVMeoF using RDMA.

For network fabrics that support RDMA, NVMeoF is able to provide an end-to-end NVMe storage solution that provides very high performance. While commands and completions are sent as "capsules", data is transferred using the fabric's RDMA mechanism. For small reads and writes, data can be bundled with the capsule, adding as little as 10 µs latency end-to-end [3]. RDMA implementations include Internet Wide Area RDMA Protocol (iWARP), RDMA over Converged Ethernet (RoCE), and Infiniband (IB). Figure 2b shows how these three architectures fit into the layered OSI model.

NVMeoF drivers are comprised of two parts, the drive-side "storage target" and the client-side "host initiator". The drive-side target driver is responsible for facilitating remote access to the drive from multiple hosts, and managing the local I/O command queues. The client host-side initiator, on the other hand, is responsible for binding to a remote drive.



(a) Without Device Lending, hosts only have access to their own individual I/O resources.

(b) Hosts can pool I/O resources together using Device Lending.

Figure 3: With Device Lending, devices become decoupled from the physical hosts they reside in. We can imagine this as connected hosts in a cluster contributing to a shared pool of I/O resources.

3 Device Lending

PCIe is the dominant standard for connecting hardware devices to computer systems. Due to its high bandwidth and low latency, it has also proved itself as a contender for use as a high-speed interconnection technology in compute clusters [6, 7, 8]. In such PCIe-based clusters, the host computers are interconnected using the same PCIe fabric as their internal devices.

Our Device Lending solution takes advantage of this. Through a process of temporarily "borrowing" remote devices and "lending" away local devices, Device Lending is mechanism for decoupling devices from the hosts they physically reside in, as illustrated in Figure 3. To the local system, the remote device appears to be dynamically hot-added, allowing local applications and device drivers to use the device transparently, without being aware that the device is actually remote.

3.1 PCIe overview

PCIe is a high-speed serial computer expansion bus standard. It uses point-to-point links, where a link consists of 1 up to 16 lanes. Each lane is a full-duplex serial connection. Data is striped across multiple lanes, so broader links yield higher bandwidth.

The internal PCIe fabric is structured as a tree. At the top of the tree is the *root complex*, consisting of CPU cores, chipset, and memory controller. The root ports act as the bridge between the PCIe fabric and the CPU. Devices form the leaf nodes of the tree, and are known as *endpoints*. Some endpoints may support multiple functions, which appear to the system as a group of distinct devices, each with a separate set of resources. The term "device" actually refers to an individual function. An example of a multi-function device is a multi-port Ethernet adapter, where individual ports can be implemented as separate functions.

Endpoints are mapped into the same address space as system memory, as illustrated in Figure 4. The system enumerates the PCIe device 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 for each of the device's memory regions. These

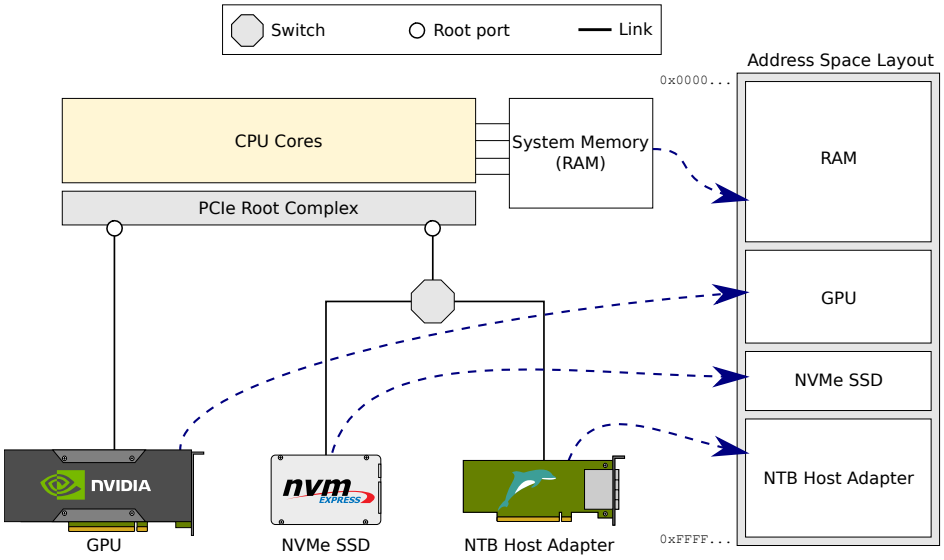


Figure 4: The system enumerates the PCIe device tree and maps device memory regions itno the same address space as system memory. Devices that are capable of DMA can access system memory as well as other devices on the PCIe fabric.

addresses are then written to the device's Base Address Regions (BARs) registers in the configuration space, making the addresses used by the system known to the device. The contents of such memory region is device specific, and is typically used for registers or exposing on-board memory.

Because this memory mapping exist, the CPU is able to read from and write to device memory just as it would access system memory. Similarly, an endpoint capable of DMA, it can read from and write to system memory. Memory accesses, i.e.. reads and writes, are forwarded over the PCIe fabric as *transactions*, and these transactions are routed based on their memory address.

Some fabrics may have switches in them, forming subtrees in the network. During the enumeration, these switches are assigned the combined address range of their downstream endpoints. This allows memory transactions to be routed hierarchically in the PCIe tree; transactions are routed 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 [5, 9]. In Figure 4, the NVMe drive and the adapter card are connected to a switch, allowing transactions between them to take the shortest path.

Finally, PCIe specifies the use of Message-Signalled Interrupts (MSIs) instead of physical interrupt lines. MSI-capable devices post a memory write to the CPU using a specific address, determined by the system during the PCIe tree enumeration. The CPU raises a interrupt specified by the contents of the write. MSI-X is an extension to MSI which supports more than a single address. This is allows targetting a specific CPU on multi-core systems.

3.2 Interconnected PCIe fabrics using NTBs

Devices are normally only capable of being part of a single PCIe fabric (root complex) at the time. As each system will enumerate their own PCIe tree and reserve BARs independently, different PCIe root complexes

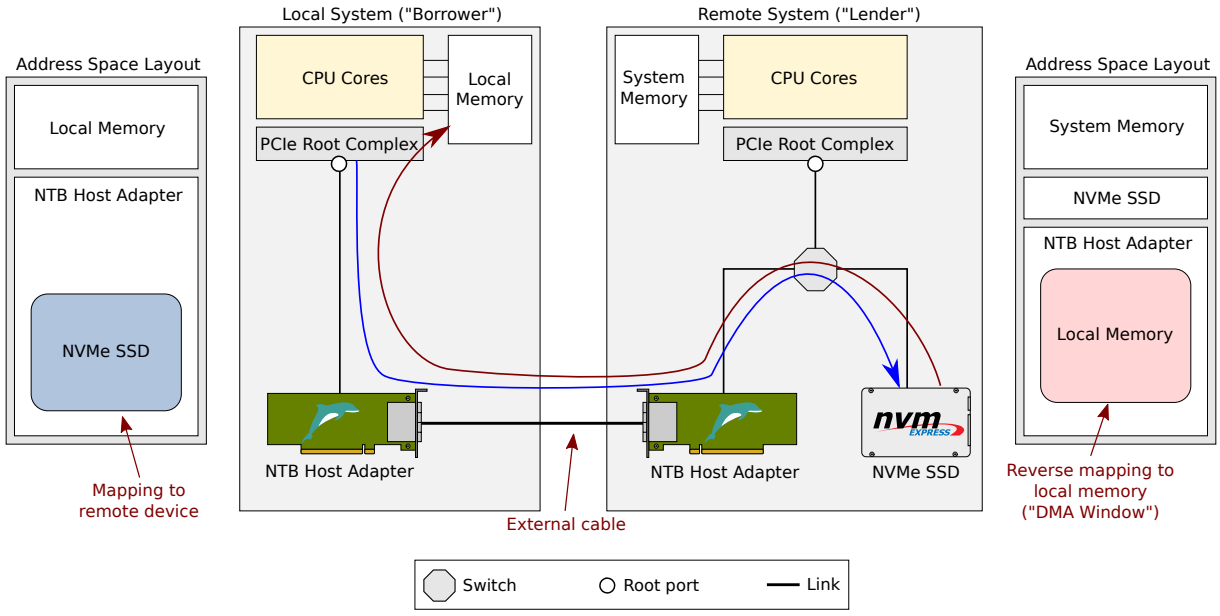


Figure 5: By mapping a remote device's memory over an NTB, a local CPU may read from and write to it. Likewise, local system memory can be mapped for the remote device, allowing it to access local memory using DMA.

will have separate address space layouts.

Non-Transparent Bridges (NTBs) are a widely adopted solution for interconnecting independent PCIe device trees [7, 10, 11, 12]. An NTB appear to a system as a regular endpoint, having one or more BARs that are reserved and mapped by the system during the PCIe device tree enumeration (also illustrated in Figure 4). As these memory ranges appear the system as any other memory-mapped device memory region, a local CPU can read and write to them. The difference from other endpoints, however, is that these ranges are not actually backed by memory. Instead, when accessing this address range, the NTB will forward transactions from one side to the other, translating addresses in the process.

Dolphin's NTB technology divides the NTB's address range into segments, which can be mapped anywhere into remote memory. A process running on the local system can map a remote memory segment into its virtual address space, allowing a program to access remote memory directly with normal memory reads and writes. This effectively creates a partitioned and distributed shared-memory architecture, without having to rely on message-passing semantics or global arrays.

3.3 Composable I/O infrastructure with Device Lending

As illustrated in Figure 5, it is possible to map the BARs of a remote device over an NTB. This allows a local CPU to access device memory across the NTB. Conversely, it is also possible to map local resources for the remote device, allowing it to write MSI interrupts and access memory on the local system across the NTB.

In order to make such mappings transparent to both devices and their drivers, we have implemented Device Lending [4, 5] for an otherwise unmodified Linux kernel using Dolphin's NTB technology. Our implementation is composed of two parts, namely a "lender", allowing a remote unit to use its device, and the

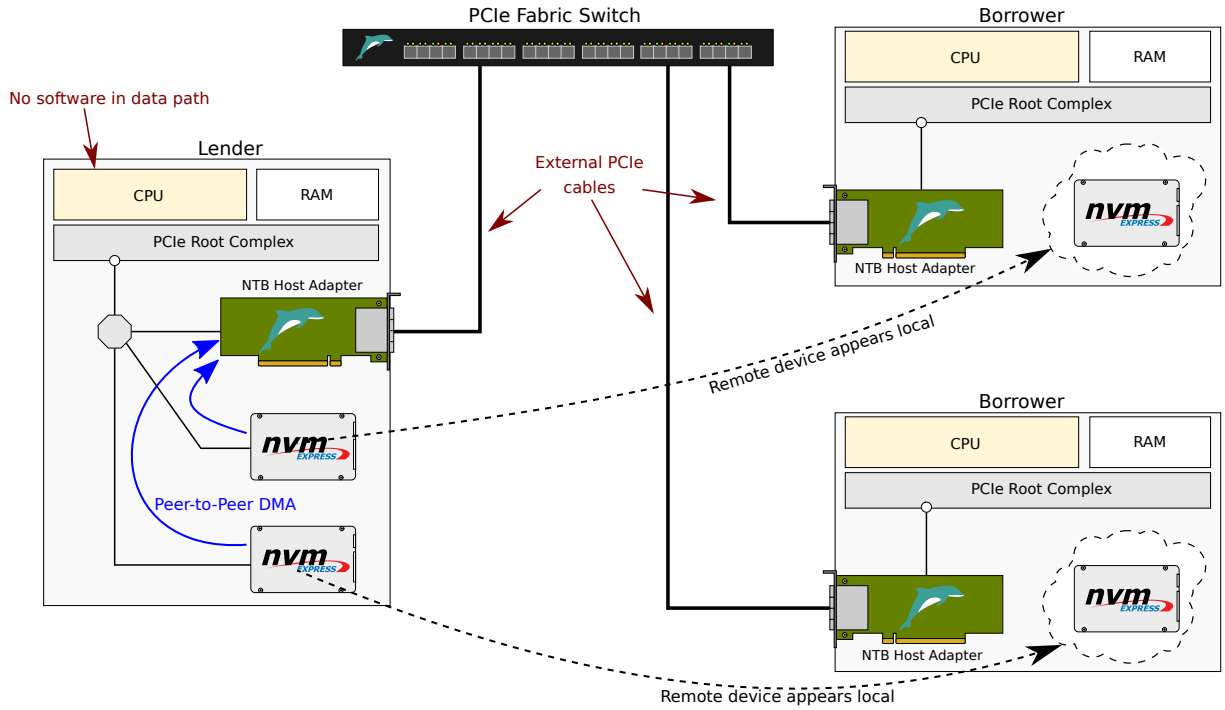


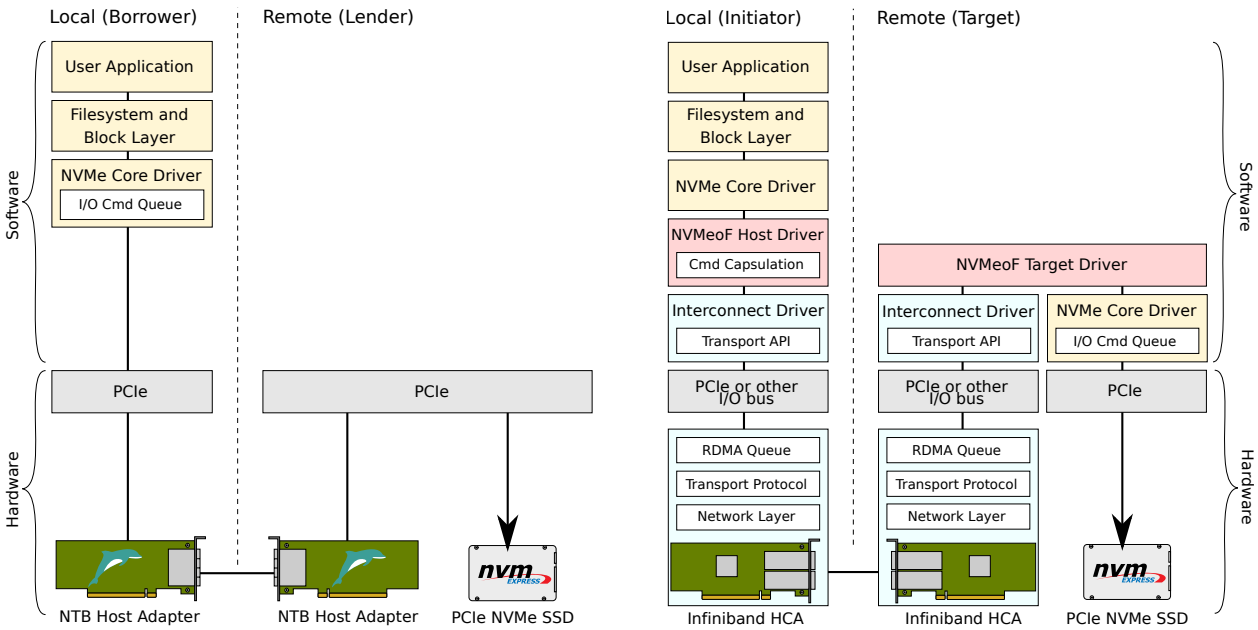
Figure 6: The Device Lending mechanism uses memory mappings to make remote devices appear as if they are installed locally. Since devices can be borrowed and returned dynamically, Device Lending is a highly flexible approach to composing the I/O infrastructure in PCIe clusters.

"borrower" using the device. By emulating a PCIe hot-plug event while the system is running, we insert a virtual device into the borrower's local device tree, making it appear to the system and device driver as if a device was hot-added in the system. The device's BARs are mapped through the NTB, allowing the local driver to read and write to device registers without being aware that the device is actually remote.

The lender is responsible for setting up reverse mappings for DMA and MSI/MSI-X. 'However, as it is generally not possible to know in advance what memory addresses the local driver might use for DMA transfers, we use the I/O Memory Management Unit (IOMMU) on the borrower to set up dynamic mappings to arbitrary adresses. This allows the lender to set up a single linear address range mapping across the NTB

When the device driver on the borrowing system calls the Linux DMA API in order to create a DMA buffer, the borrower injects the remote I/O address prepared by the lender and sets up a local IOMMU mapping to the allocated DMA buffer. From the drivers point of view, this I/O address is no different than any local I/O address, and it passes it to the device, completely unaware that the address is a remote-side address. All address translations between the different address domains are done in hardware, which has very low latency.

The Device Lending method is a highly flexible method of sharing devices in a PCIe cluster and composing custom I/O architectures, as illustrated in Figure 6. By allowing remote devices to appear to a system as if they are locally installed, Device Lending is a method for decoupling devices from the systems they physically reside in. Hosts can act as both lender and borrower, and devices can be temporarily assigned and reassigned dynamically.



(a) Remote NVMe access using Device Lending.

(b) Remote NVMe access over IB using NVMeoF.

Figure 7: Device Lending makes the remote NVMe drive appear local, eliminating the need for forwarding I/O commands and RDMA transactions to the remote system. Instead, the local NVMe driver can access the drive directly over PCIe.

4 NVMe over PCIe fabrics

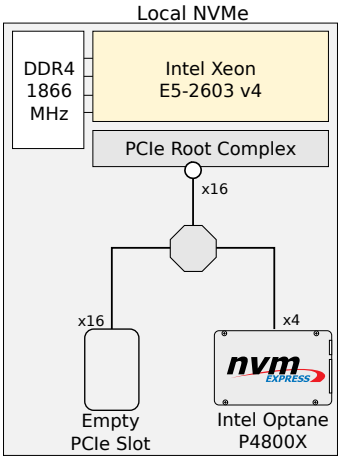
In a Dolphin PCIe network, hosts and their internal devices are interconnected to the same PCIe fabric. Using the Device Lending method, remote devices can be mapped into a local system's address space, appearing to the system as if it is locally installed. A remote NVMe drive can be borrowed by the system, and used directly by a local driver, without the need for forwarding I/O commands and RDMA transactions. Even though NVMeoF using RDMA is extremely efficient, there is some inevitable software overhead in encapsulating, forwarding and receiving I/O commands, as well as the added latency from translating read and write requests into fabric-specific RDMA messages. The difference between Device Lending and NVMeoF using IB is shown in Figure 7.

4.1 Kernel NVMeoF driver benchmark

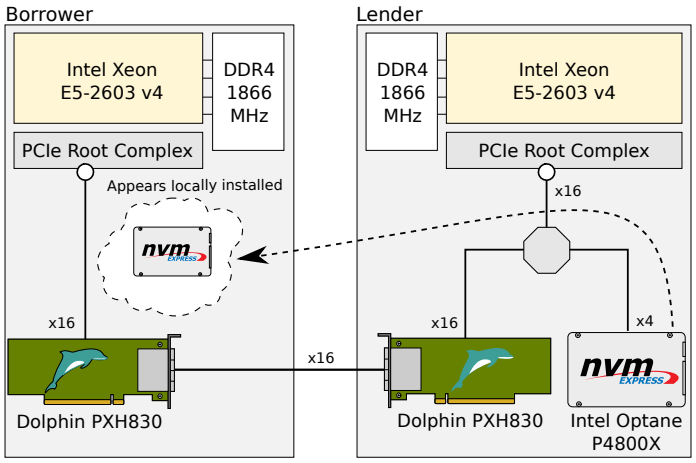
In order to evaluate impact on NVMeoF using RDMA on access latency, we have benchmarking experiments for three scenarios:

- A host using a local NVMe drive.
- A host using our Device Lending mechanism to access a remote NVMe drive.
- A host using the Linux kernel NVMeoF using RDMA to access a remote drive over IB.

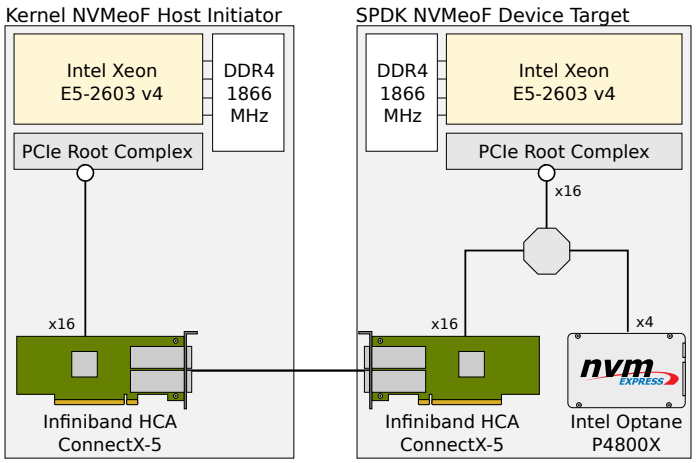
Figure 8 shows the hardware configuration used in the different scenarios. An Intel Optane P4800X NVMe drive was connected to a host via an expansion chassis. This is illustrated in a simplified way in Figure 8a. The transparent bridge between the host and the expansion chassis is a PCIe Gen3 x16 link, using Dolphin's MXH832 host adapter and MXH833 target. The drive is PCIe Gen3 x4. We configured version 3.13 of



(a) Accessing a local drive using the Linux kernel NVMe driver.



(b) Accessing a remote drive using Device Lending. As the drive appears local, the borrower uses the standard Linux NVMe driver to access it.



(c) Accessing a remote drive over IB using the kernel NVMeoF using RDMA driver.

Figure 8: Simplified block diagram of the hardware configurations and topologies used in our benchmarking.

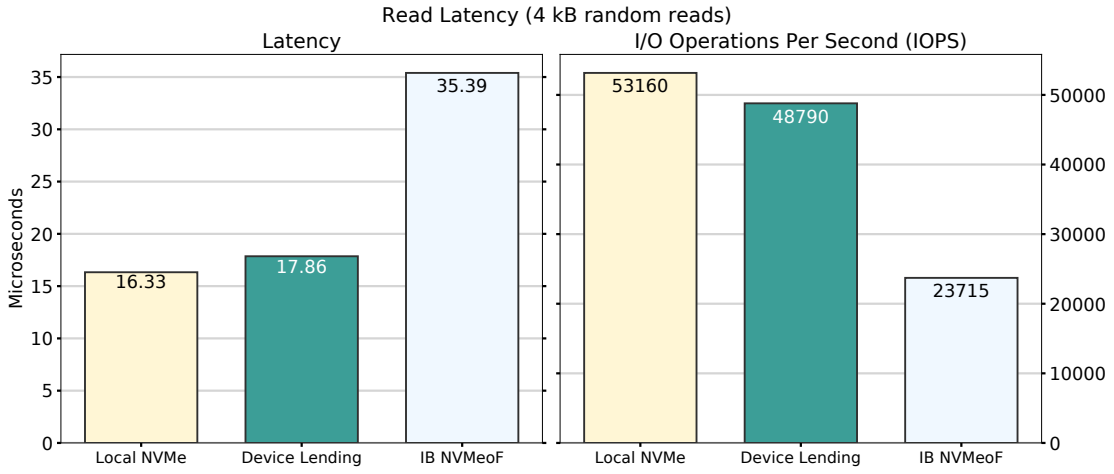


Figure 9: Results of our NVMeoF access latency benchmark using the Linux kernel driver stack. The benchmarking program executed 327,680 reads of 4 kB each, using a random pattern.

Flexible I/O Tester (FIO) on an Ubuntu 18.04.2 LTS installation using version 4.15 of the Linux kernel to do a series of 4 kB reads in a random pattern.[1] This access pattern was chosen as every single read requires issuing a separate I/O command.

In the Device Lending scenario, shown in Figure 8b, a PXH830 NTB host adapter card was placed in the empty PCIe slot next to the NVMe drive in the expansion chassis. Another host, the "borrower" (also running Ubuntu), was connected by an external PCIe Gen3 x16 cable. This host was configured to "borrow" the drive from the remote system, using the standard Linux NVMe driver to access it over the external PCIe link.

Finally, in the NVMeoF over IB scenario, we connected two hosts using an IB a IB ConnectX-5 host card adapter on each side. On the initiator-side, we used the standard Linux NVMeoF driver, configured to use RDMA. The drive-side host was configured to run version 19.1.1 of Storage Performance Development Kit (SPDK) as the target driver, also using RDMA.

Figure 9 shows the 99th percentiles of the read latencies for our benchmarking tests as well as the number of IOPS for the same tests. Note that 4 kB reads in a random pattern is not an access pattern optimized for maximizing throughput, but rather serves as a useful test for measuring I/O command overhead. We observe that Device Lending is almost able to achieve local performance, and the added latency comes from having to traverse the external PCIe link. This particularly impacts DMA reads, such as when the drive is fetching I/O commands. In the NVMeoF scenario, however, the read operations are affected by having to be encapsulated into messages, transmitted over the interconnect, and interpreted by software on the other end.

4.2 Advanced storage applications

SPDK [13] is a popular software library for developing custom high-performance and scalable storage applications. By relying on drivers implemented in user-space, it is able to by-pass the Linux file system and block drivers, and access a drive directly from an application. As part of the framework, SPDK also implements a NVMeoF using RDMA stack which aims to provide very low end-to-end latency.

[1] FIO uses the Linux Asynchronous I/O mechanism (AIO) for reading from disk.

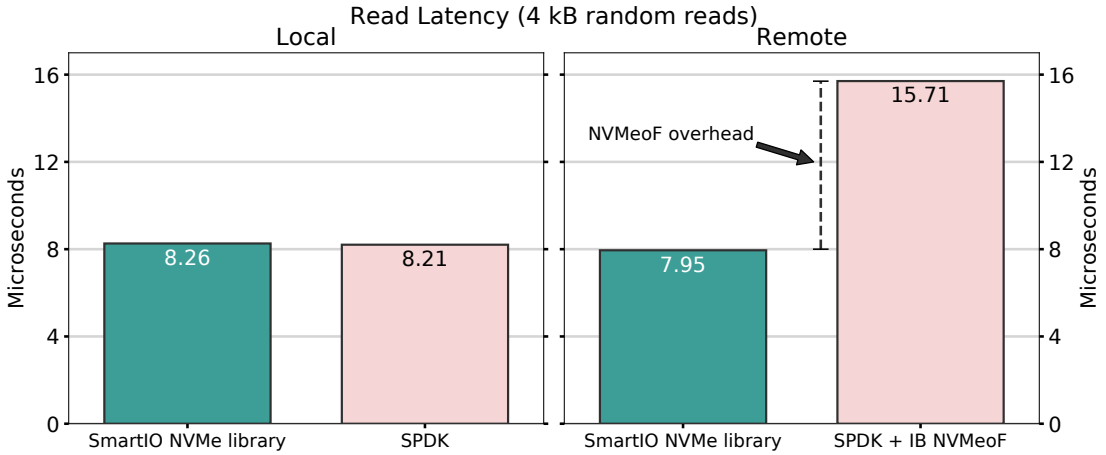


Figure 10: Results of our NVMeoF access latency comparison between SPDK and our SmartIO NVMe library. The FIO benchmarking program executed 327,680 reads of 4 kB each, using a random pattern. Note the overhead added by the NVMeoF protocol.

As part of Dolphin's SmartIO technology, we have also extended the SISCI API with device-oriented Device Lending semantics. This allows a software developer to implement a user-space device driver that supports distributed operation. Using the SmartIO extension, Dolphin has developed a NVMe software library for providing distributed access to NVMe drives in a PCIe cluster [14].

Using the same hardware configurations as shown in the previous section (Figure 8), we have compared SPDK to our SmartIO NVMe library accessing a local drive. We also compared SPDK using NVMeoF using RDMA over IB to our library accessing a remote drive, highlighting the impact on latency that the NVMeoF stack adds. As in the previous section, we used FIO to conduct a series of 4 kB reads in a random pattern. For the local case, SPDK was configured to access the drive using directly. For the remote case, we used SPDK as the device-side NVMeoF using RDMA target driver over IB.

The results are shown in Figure 10, where we have plotted the 99th percentiles of the read latencies. NVMeoF using RDMA adds approximately 7.5 µs compared to SPDK running locally. It is interesting to note that our SmartIO library actually has lower latency accessing the disk remotely compared to locally. This is because we are using slightly different PCIe switch technology in our transparent- and NTB cards in this set up.[2] The device accessing memory using DMA over the links adds an accumulated latency that is slightly higher in the local transparent configuration.

As this benchmark uses SPDK on both sides accessing the same NVMe drive using the same hardware, we can isolate the overhead caused by the NVMeoF using RDMA protocol alone. While it achieves its goal of not adding any more than 10 µs end-to-end, it is no match compared to DMA over PCIe.

[2]The PXH830 NTB host adapter uses Broadcom/PLX chips, while the MXH832/MXH833 transparent adapter card uses chips from Microsemi.

5 Summary

In this paper, we have explained the Device Lending mechanism and how it can be used to access remote NVMe drives in a PCIe cluster. We have also compared Device Lending to NVMeoF using RDMA, particularly the difference between Device Lending and accessing a remote drive over IB. We have presented our latency benchmarking results, comparing Device Lending to NVMeoF using IB for both the Linux kernel implementation and SPDK implementation of NVMeoF using RDMA.

Disclaimer

Notes

This document is based on information available at the time of publication. While efforts have been made to be accurate, the information contained herein does not purport to cover all details or variations in hardware and software.

References

[1] *NVM Express Base Specification*. Standard. Revision 1.4. NVM Express Inc. June 2019.

[2] *NVMe Overview*. NVM Express Inc. URL: https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf (visited on 07/14/2019).

[3] *NVMe over Fabrics Overview*. NVM Express Inc. 2015. URL: https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf (visited on 07/14/2019).

[4] L. B. Kristiansen, J. Markussen, H. K. Stensland, M. Riegler, H. Kohmann, F. Seifert, R. Nordstrøm, C. Griwodz, and P. Halvorsen. "Device Lending in PCI Express Networks". In: *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV. 2016, 10:1–10:6. DOI: 10.1145/2910642.2910650.

[5] J. Markussen, L. B. Kristiansen, H. K. Stensland, F. Seifert, C. Griwodz, and P. Halvorsen. "Flexible Device Sharing in PCIe Clusters using Device Lending". In: *Proceedings of the International Conference on Parallel Processing Companion*. ICPP Companion. 2018, 48:1–48:10. DOI: 10.1145/3229710.3229759.

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

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

[8] M. Ravindran. "Extending Cabled PCI Express to Connect Devices with Independent PCI Domains". In: *Proceedings of the IEEE Systems Conference*. SysCon. 2008, pp. 1–7. DOI: 10.1109/SYSTEMS.2008.4519048.

[9] *Remote Peer to Peer made easy*. White paper. Dolphin Interconnect Solutions. Aug. 2015.

[10] C.-C. Tu and T.-c. Chiueh. "Seamless Fail-over for PCIe Switched Networks". In: *Proceedings of the International Systems and Storage Conference*. SYSTOR. 2018, pp. 101–111. DOI: 10.1145/3211890.3211895.

[11] A. Kamzi. "PCI Express and Non-Transparent Bridging support High Availability". In: *Embedded Computing Design* (2004).

[12] PLX Technology. 2005. URL: https://docs.broadcom.com/docs-and-downloads/pdf/technical/expresslane/NTB_Brief_April-05.pdf.

[13] *Storage Performance Development Kit*. URL: https://github.com/spdk/spdk/tree/v19.01.1 (visited on 03/11/2019).

[14] J. Markussen. *libnvm: An API for building userspace NVMe drivers and storage applications*. Dolphin Interconnect Solutions. URL: https://github.com/enfiskutensykkel/ssd-gpu-dma/tree/sisci-5.11 (visited on 08/02/2019).