Multi-Host Sharing of a Single-Function NVMe Device in a PCIe Cluster

Jonas Markussen
Dolphin Interconnect Solutions
jonas@dolphinics.com

Lars Bjørlykke Kristiansen
Dolphin Interconnect Solutions
larsbk@dolphinics.com

Håkon Kvale Stensland
Simula Research Laboratory
University of Oslo
haakonks@simula.no

Pål Halvorsen
SimulaMet
Oslo Metropolitan University
paalh@simula.no

***Abstract*—Distributed cluster applications, including machine learning tasks, database applications, and HPC workloads, often rely on NVMe-oF using RDMA for fast, block-level access to storage devices over a network. However, RDMA solutions add extra latency by requiring software on the critical path. In this paper, we present a distributed NVMe driver for sharing NVMe storage devices across hosts in a PCIe cluster. By building on PCIe shared memory capabilities, we demonstrate disaggregation of NVMe controllers at the I/O queue level, allowing them to be used in parallel by remote hosts without relying on RDMA. Our experimental results prove that our PCIe-based solution reduces network latency and is comparable to local access.**

***Index Terms*—PCIe, NVMe, Storage, HPC**

I. Introduction

Non-Volatile Memory Express (NVMe) [1] is an interface specification for storage device controllers attached to the PCIe bus, most commonly used for solid-state flash memory drives (SSDs). Compared to traditional spinning hard disks, SSDs have lower latency and can support a higher degree of parallelism. This parallelism is reflected in the design of NVMe by relying on the inherent memory addressing capabilities of PCIe devices, i.e., Direct Memory Access (DMA), to support multiple independent I/O queues that operate in parallel. By avoiding any form of locking or software synchronization in the command submission and completion paths, NVMe devices can perform I/O operations with very high throughput and low latency.

In recent years, we have seen a convergence of high-performance computing, big data, and machine learning problem areas, making distributed, high-volume storage a requirement for workloads running on networked hosts in a computing cluster [2], [3]. As NVMe SSDs are significantly faster than traditional storage devices, the performance bottleneck in networked storage solutions is no longer the storage devices but the network itself. To address this, NVMe over Fabrics (NVMe-oF) has emerged as the industry standard for accessing an NVMe controller over a network ("fabric") [4]. By using Remote DMA (RDMA) zero-copy transfers, NVMe-oF implementations extend the parallel design of NVMe over a network. Individual I/O queues are "bound" to remote hosts, allowing clients to enqueue commands directly in memory on the server using RDMA. Thus, NVMe-oF using RDMA enables direct access to remote storage devices at the block level and eliminates almost all communication latency. However, as RDMA solutions are implemented at the application level, software is still required to operate the server's NVMe controller. This inevitably leads to additional latency because software is required in the critical path.

In this paper, we extend our previous work on distributed storage in PCIe-networked clusters without relying on RDMA [5]. The Software Infrastructure Shared-Memory Interconnect API (SISCI) [6] provides host-to-host shared-memory communication in Dolphin PCIe clusters. This API has been extended with device-oriented functionality, providing capabilities for mapping a shared global address space for a PCIe device and allowing devices to use native DMA to access shared memory regions. Using the functionality provided by our extension, we have built a proof-of-concept kernel space NVMe driver that combines storage I/O with shared-memory functionality. We demonstrate how a single NVMe controller may be shared in parallel by multiple hosts, by using shared-memory regions to distribute I/O queues to remote hosts.

To demonstrate the strength of our approach, we have conducted performance benchmarks using synthetic storage workloads, demonstrating the latency benefits of native PCIe compared to NVMe-oF. Whereas state-of-the-art NVMe-oF implementations using RDMA can achieve bandwidth comparable to local performance, they can increase access latency by several microseconds (µs). In contrast, our experimental results prove that by avoiding software in the path and relying on native PCIe, network access latency is almost eliminated, and our solution is comparable to that of accessing a *local* NVMe device. In short, this paper makes the following contributions:

- We have implemented a proof-of-concept, distributed kernel space NVMe driver using shared memory communication. Our driver allows a single NVMe controller to be operated in parallel by multiple remote hosts, providing them with direct block-level access.
- We provide an evaluation of the latency benefits of our PCIe-based shared memory approach through a comparative performance experiment using synthetic workloads. Compared to NVMe-oF using RDMA, our solution has minimal network access latency.

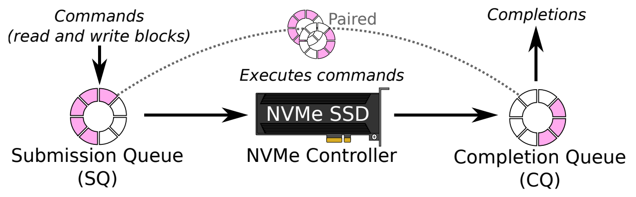


Fig. 1: NVMe supports asynchronous operation by using queues for the command submission and completion paths.

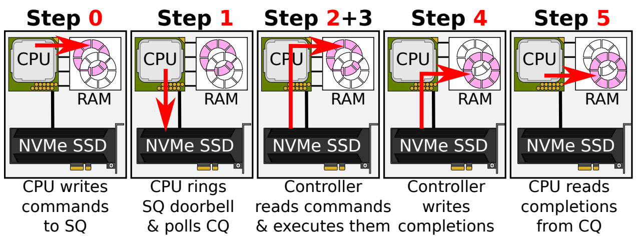

Step 0
Step 1
Step 2+3
Step 4
Step 5
CPU
RAM
NVMe SSD
CPU writes commands to SQ
CPU rings SQ doorbell & polls CQ
Controller reads commands & executes them
Controller writes completions
CPU reads completions from CQ

Fig. 2: NVMe operation in a local host (left to right).

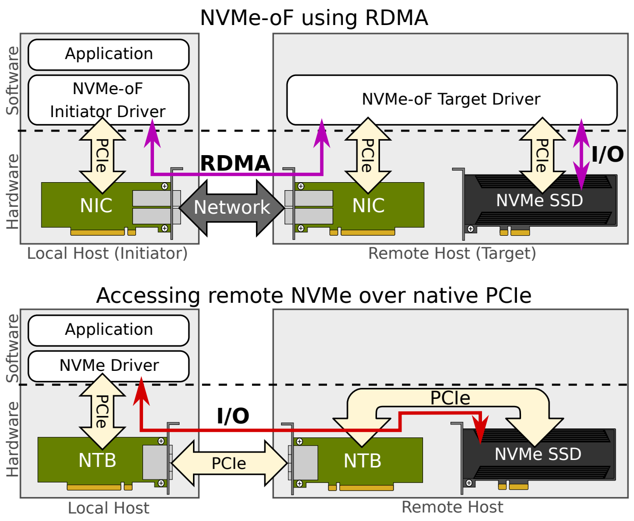


Fig. 3: Accessing remote storage using NVMe-oF vs. PCIe.

II. Background and Motivation

An NVMe controller is associated with a single PCIe device function [1]. Some NVMe SSDs may support multiple PCIe device functions or implement Single-Root I/O Virtualization (SR-IOV) [7] where the device appears to have multiple (virtual) functions. Each virtual or physical function has its own separate set of PCIe resources and memory regions and implements a stand-alone NVMe controller with access to the same underlying storage medium. However, due to the complexity of implementing multiple device functions (either physical or virtual) in hardware, few NVMe devices on the market support this.

The NVMe standard [1] enables highly parallel operation by using separate, asynchronous I/O queues for the command submission and completion paths. As illustrated in Figure 1, one or more Submission Queues (SQ) are associated (paired) with a Completion Queue (CQ), allocated and configured by driver software. Queues are implemented as ring buffers and can be allocated anywhere in physical memory, entirely at the discretion of the NVMe controller's driver. Each queue has an associated doorbell register, which driver software uses to notify the controller. Figure 2 shows the basic operation of an NVMe controller: The driver enqueues commands in an SQ, for example, reading N number of blocks from storage to a specified location in memory. It then notifies the NVMe controller that commands are enqueued by "ringing" the SQ's doorbell register, and waits for corresponding completions to be posted in the associated CQ by polling CQ memory or waiting for a device-generated interrupt. When notified about entries in the SQ, the controller fetches commands using DMA, executes them, and then posts completions to the CQ.

NVMe-oF using RDMA [8] extends this parallelism over an InfiniBand network. NVMe-oF implementations are composed of 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 initiator. Similar to the SQ and CQ pairing mechanism, the underlying InfiniBand transport layer also uses pairs of Work Queues and Completion Queues. InfiniBand-capable network adapters allow queues hosted in system memory, similar to NVMe, which allows InfiniBand applications to post-work requests, , i.e., RDMA Send and Receive, and poll for completions directly in memory. Additionally, this design maps very well onto NVMe-oF architecture; the NVMe-oF target driver can "bind" a Receive Work Queue to an NVMe SQ, allowing I/O commands to be enqueued directly in memory on the target. Therefore, the target driver can start I/O operations as soon as commands are enqueued without requiring any processing. Similarly, the NVMe CQ can be "bound" to a Send Work Queue, allowing sending back completions to the initiator as soon as the I/O operation completes. However, while RDMA enables efficient, one-sided initiation of I/O operations over a network, an inevitable latency penalty stems from involving software in the data path (as illustrated in Figure 3). Some network adapters implement an NVMe-oF target driver in hardware or firmware (target offloading) to reduce CPU load and latency. Another approach would be directly implementing RDMA capabilities into device controllers, as proposed by Daglis et al. [9]. However, this increases the complexity of hardware implementations and, thus, limits their availability.

By using a special type of PCIe device called Non-Transparent Bridge (NTB), separate computer systems can be connected to the same shared PCIe network [5], [10], [11]. NTBs can be embedded as a CPU feature [12], but are more commonly implemented in PCIe switch chips [13], allowing independent computer systems to interconnect with plug-in host adapter cards and external cables [14], [15]. The defining feature of NTBs is that arbitrary memory address translations can be configured, allowing segments of remote memory to be

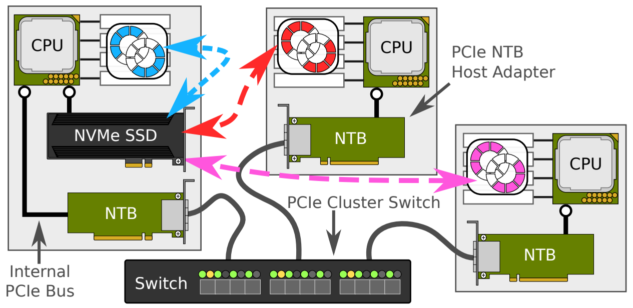


Fig. 4: Multiple hosts can operate the same NVMe controller by distributing I/O queue pairs in a PCIe cluster.

memory-mapped for a local system, thus providing "windows" into a remote system's address space.

In the context of NVMe, we have previously built a proof-of-concept implementation, showing that it is possible to configure queues' memory addresses through such "NTB windows" and allow controllers to access remote memory directly using native DMA [5]. Similarly, by mapping the PCIe device function associated with the NVMe controller, each host has access to device registers. Figure 4 shows how to distribute queues in such PCIe clusters, allowing multiple remote hosts to operate a single controller in parallel. Queues do not require any form of synchronization and can be distributed to remote hosts and used to operate the device independently. The NVMe controller can fetch commands from the different SQs, post completions to CQs, and load and store data in buffers using native DMA.

III. PCIe Shared-Memory Clustering

PCIe [16] is the most widely used I/O bus standard for connecting devices to a computer system. The defining feature of PCIe is that devices are mapped into the same address space as the CPU. When a system boots up, it enumerates all the devices attached to the PCIe fabric. The system assigns a memory address range for each device's memory region and writes them to its Base Address Register (BAR). As such, "BAR" is used synonymously for individual regions of device memory. Because this mapping exists, a CPU can read and write to device memory the same way it would access system memory, and it is called memory-mapped I/O (to distinguish it from legacy port-based I/O). Likewise, if a device is capable of DMA, it can read from and write to system memory. A device may even access other devices on the fabric, as they are mapped into the same address space. PCIe switches may also be used in the fabric; memory transactions are routed in the fabric based on their addresses, and since switch ports are assigned the combined address range of their downstream devices, memory transactions can be routed shortest path ("peer-to-peer").

Individual systems have separate PCIe address spaces, but an NTB makes it possible to connect systems over PCIe (Figure 5). An NTB appears as a regular device with associated BARs that are assigned address ranges by the local system. However, instead of being backed by memory or device registers, reads and writes to these BARs are forwarded from one side of the NTB to the other, translating the memory addresses in the process and enabling access to remote memory as if it were local device memory. As illustrated in Figure 5, by dividing the BAR into several ranges and using a different base offset per range, it becomes possible to map arbitrary memory regions in a remote system. The local address is stored with a far-side address mapping in a look-up table on the NTB, thus providing "windows" into remote address space. This allows hosts to map parts of a remote host's memory through their local NTB using different base offsets.

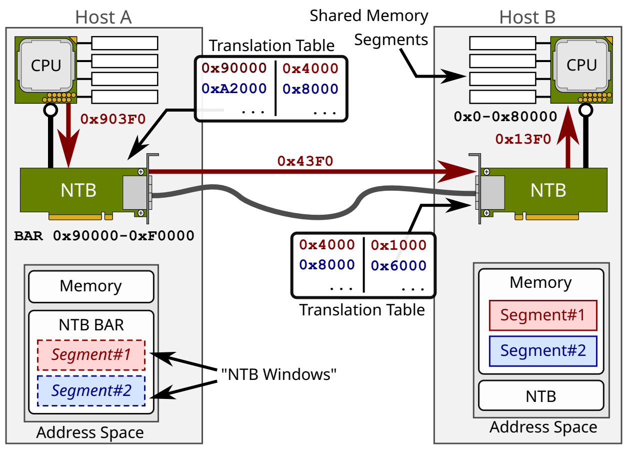


Fig. 5: Two independent systems connected with NTBs.

IV. Shared-Memory API Extension

By mapping remote memory regions into local address space, we can create a heterogeneous PCIe-networked cluster using NTB adapter cards and NTB-capable cluster switches. Multiple hosts may map the same parts of memory through their respective NTB adapters, creating a global address space shared by PCIe-interconnected hosts.

Using the SISCI API [6], applications are provided with a high-level interface for configuring PCIe NTBs, thus enabling shared-memory functionality for application software running in such networks. Hosts may allocate linear contiguous regions in physical system memory, called "segments". Other hosts can connect to these segments and map them through their own local NTBs. The physical address range of these NTB mappings can then be memory-mapped into the virtual address space on the local host, effectively allowing software to read and write to remote memory the same way they would read and write to local system memory. Software running on different hosts may map the same memory segments, thus implementing distributed shared-memory applications.

We have extended the SISCI API with device-oriented functionality. Figure 6 illustrates the main functionality of our extension, specifically how device registers can be mapped for a local CPU and how a shared memory segment is mapped for a device. By exposing device BARs as shared memory

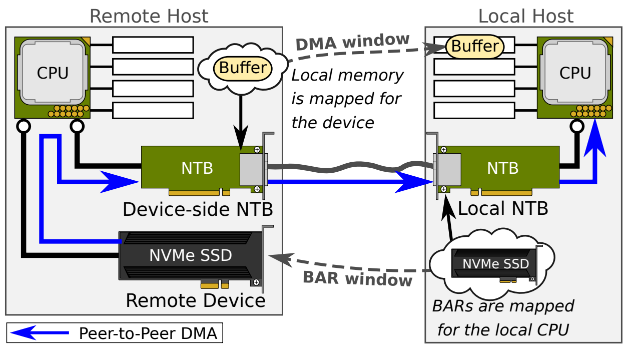


Fig. 6: Our SISCI API extension provides functionality for mapping remote, shared memory regions for a device, and memory-mapping device registers as a shared memory region.

segments, a process running on the local CPU can memory-map the device registers of a remote device through the CPU-side NTB. Likewise, by setting up mappings to shared memory segments on a device-side NTB, a device can access remote memory using its native DMA capabilities. To support this, we have implemented the following functionality:

- A host abstraction service called SmartIO runs on all hosts in the cluster. In SISCI, the application must be aware of which host memory segments they reside in to memory-map them through the local NTB. SmartIO allows a user to register devices for use with our API extension and assigns them a unique, cluster-wide identifier. Software running anywhere in the network can uniquely identify devices regardless of which physical hosts they are installed in. Devices that are registered with our service automatically have their BARs exported as SISCI segments, allowing them to be memory-mapped into virtual address space. Moreover, our SmartIO service tracks devices and which hosts they reside in, SmartIO supports translating device-side physical addresses in the different hosts, including device-side NTB mappings to shared memory segments. This removes the complexity of dealing with multiple physical address spaces in different hosts, for example, when initiating a DMA transfer.
- API calls for dynamically acquiring and releasing a device reference from a process. Acquiring a device reference can either be exclusive, allowing only a single process to manage the device at the time, or non-exclusive, allowing several application instances to access the device simultaneously. A single instance can first lock the device to reset, initiate, and prepare the device before allowing others to access the device. Furthermore, as our SmartIO abstraction service distributes information about devices to other hosts in the network, we also support dynamically discovering devices registered with SmartIO.
- API calls for mapping SISCI segments on behalf of a (remote) device, effectively setting up mappings over the device-side NTB to memory regions. Segments can be either local or remote to the device. We call such mappings "DMA windows" as mapping SISCI segments for a device allows it to use native DMA to read and write to shared memory regions. Our SmartIO abstraction service will automatically resolve device-side physical addresses to (remote) memory segments under the hood, allowing the same software to run on any cluster host and remain agnostic about the specific address space layout in other hosts.
- API calls 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 our SmartIO service choose which host to allocate memory is 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.

Our SISCI API extension effectively provides an abstraction layer between hosts and their resources, , i.e., their devices and memory. Consequently, our extension makes it possible to implement a device driver that can run on any host in the network, operating a remote device anywhere in the cluster. The complexity of dealing with multiple (and different) physical addresses space layouts are removed by providing API calls to resolve memory addresses to remote memory segments for a device. This allows a host to read and write to registers and initiate native DMA transfers to shared memory segments without being aware of the specific PCIe topology.

V. Distributed NVMe Driver Implementation

We have extended our previous proof-of-concept, user-space NVMe driver [5] by implementing it as a kernel space module instead of providing a shared NVMe device as a block device in Linux. One of the motivations for implementing a block device driver was to use shared disk file systems available on Linux, such as Global File System (GFS) or Oracle Cluster File System (OCFS). As mentioned in Section II, NVMe uses a mechanism for paired submission and completion queues. Both SQs and CQs are allocated in memory configured by the device driver software, and the controller uses DMA to both fetch commands and post completions. Each queue has its dedicated doorbell register, which driver software writes to, to notify the controller about new queue entries. This allows driver software to operate queues in parallel and interact with the controller without any form of locking or contention.

Since queues can be allocated anywhere in memory, entirely at the discretion of the driver, we can use addresses that map over an NTB for software operating the device. Figure 7 shows how the device-oriented API extension described in Section IV is used to extend NVMe operation out of a single computer system, allowing multiple hosts to operate the same controller simultaneously. We use the SmartIO abstraction service to map queue memory and data buffers for the NVMe controller. As the service handles device-side address space translation,

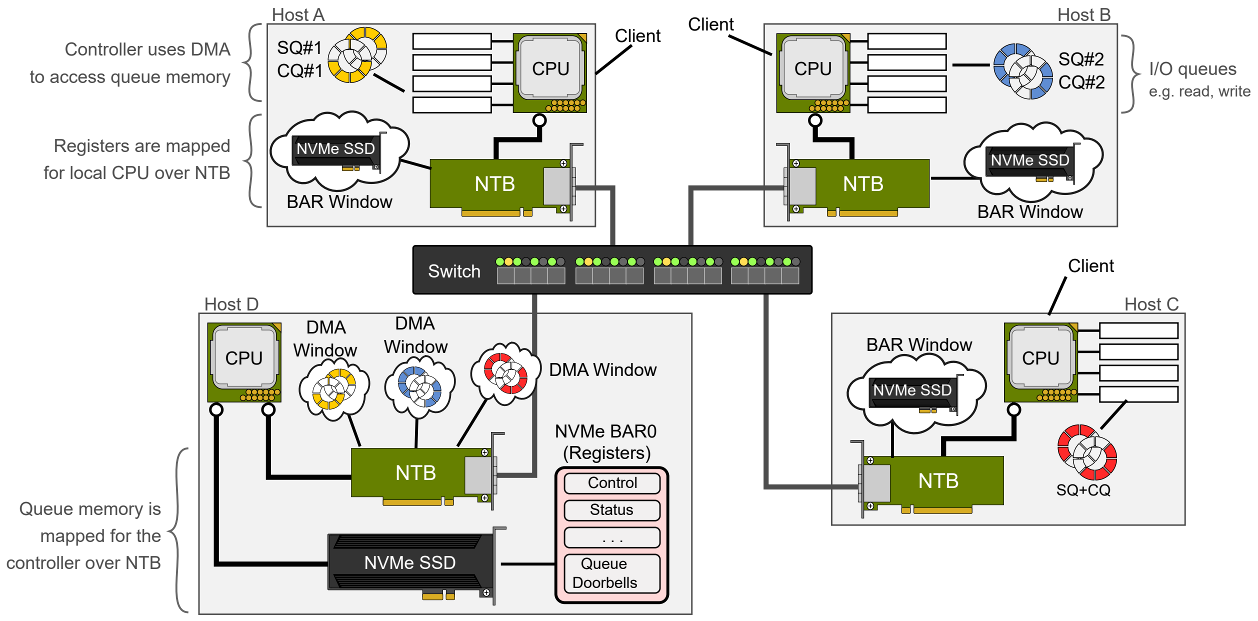


Fig. 7: Distributed driver architecture: The NVMe controller accesses queues residing in memory segments on remote hosts (DMA windows). Likewise, each host map device registers by mapping the BAR of the NVMe device through their respective NTBs (BAR windows). All hosts see the same NVMe device as a local block device and can simultaneously read and write to it.

we do not need to care about the physical address space on the host where the device resides other than using resolved addresses when setting up queues. Furthermore, as device BARs are automatically exported as shared memory segments, mapping device registers, i.e., queue doorbell registers, for a local CPU is trivial.

While several I/O queues are supported in NVMe, there can only be one pair of *admin queues*. Our implementation consists of a “manager” kernel module and one or more “client” kernel modules. The manager is responsible for initializing the controller, setting up the admin queues, and performing privileged tasks, such as creating and deleting I/O queue pairs, on behalf of the clients. The manager also allocates a shared memory segment associated with the controller with metadata about the manager, such as which host it runs on. This informs clients that the device is being managed and tells them how to contact the manager.

A client module uses one or more I/O queue pairs to read or write blocks from the NVMe controller and is also responsible for registering a block device with the operating system. The client bootstraps itself by reading the shared memory segment created by the manager and requests an I/O queue pair from the manager. Once the requested I/O queues are created, the client uses them to operate the controller independently of the manager and other clients. After creating an I/O queue pair, the client maps this queue pair with the Linux kernel block layer request queues and registers a block device.

Memory reads are non-posted transactions, as a read request requires a completion (with the requested bytes) to be returned to the requester. As such, reads are affected by the number of switch chips in the path between requester and completer; the longer the path between a device and the memory it reads from, the higher the request-completion latency becomes. This raises an issue with our implementation, as moving queue memory out of the local device memory to the memory in a different host entirely increases the distance the controller needs to read across to fetch commands. However, as the NVMe standard has no restrictions regarding memory locations for paired queues, any address a controller can use DMA to is a valid queue memory location.

Figure 8 illustrates how an SQ hosted in device-side memory can be paired with a CQ in local memory. As described in Section IV, our API extension supports specifying access pattern hints when allocating memory segments. By specifying that the CQ segment will be mostly read from the CPU and only written to by the device, the underlying SmartIO abstraction layer will prefer allocating the CQ segment to local memory. Similarly, by specifying read access by the device and only write access by the CPU for the SQ segment, the system will prefer to allocate it in device-side memory. The SQ memory segment is mapped through the local NTB, allowing the local CPU to write commands directly into device-side memory. As writes are posted transactions, the CPU can immediately “ring” the doorbell register after writing the commands. This notifies the NVMe controller, which can then read commands from local memory rather than reaching over the NTB. Once the commands are executed, the controller will write completions to memory local to the CPU. As our SISCI

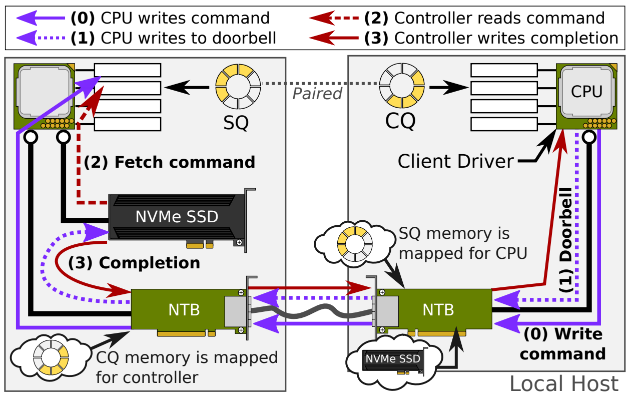

(0) CPU writes command
(1) CPU writes to doorbell
(2) Controller reads command
(3) Controller writes completion
SQ
Paired
CQ
CPU
Client Driver
(2) Fetch command
NVMe SSD
(3) Completion
SQ memory is mapped for CPU
(1) Doorbell
NTB
NTB
(0) Write command
NVMe SSD
CQ memory is mapped for controller
Local Host

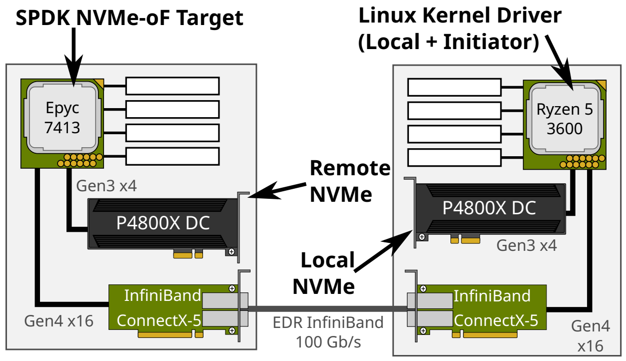
Fig. 8: Allocating the SQ in memory closer to the controller reduces the distance it needs to read across to fetch commands. SQ memory is mapped for the local CPU over the NTB, allowing it to write directly into device-side memory.

API extension does not currently support device-generated interrupts, the client driver can poll on local memory for CQ events.

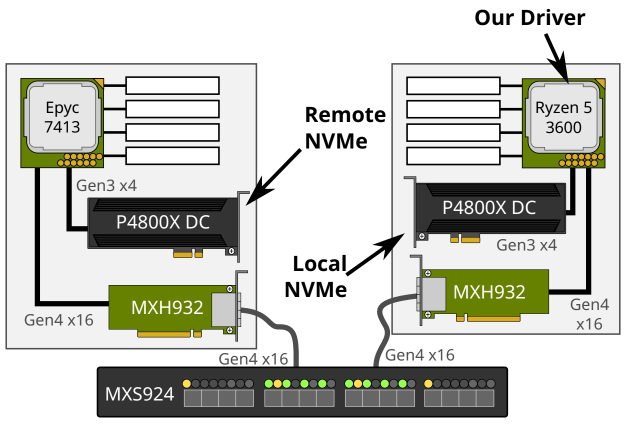
A significant difference from our previous user-space implementation [5] is that while it was possible to map SISCI shared-memory segments for the NVMe controller, allowing an application to use these as DMA buffers directly, our driver must instead handle I/O requests from the Linux block layer. An I/O request contains a pointer to an arbitrary memory buffer, where disk data should be written to or read from. Configuring NTB mappings on the fly is not a feasible solution, as this would cause a significant delay in the critical I/O path and severely limit the number of I/O requests in flight. Instead, the client driver registers a large DMA buffer segment mapped for the NVMe controller. This buffer is partitioned so that each I/O request has a dedicated, equal-sized partition of the buffer. The driver uses this DMA buffer as a *bounce buffer* when issuing NVMe I/O commands. The benefit of this approach is that NVMe DMA descriptors can be programmed once since the DMA buffer segment is constant. The downside of this approach is that an extra memory copy is needed in either the command submission path (writes) or the completion path (reads). A future extension of the NVMe driver is to use the I/O Memory Management Unit (IOMMU) to dynamically map buffer addresses for each request instead of using a bounce buffer.

VI. Evaluation

The performance advantage of our implementation is that it provides direct access to remote NVMe devices over native PCIe. However, by using modern networking technologies, such as InfiniBand and 100/200 Gb/s Ethernet, remote storage solutions like NVMe-oF using RDMA can provide very high throughput, which is comparable to that of local PCIe [8]. As such, our evaluation compares our driver implementation to NVMe-oF using RDMA. Our presented experiments focus on latency measurements, as we argue this is where the main



(a) Using the stock Linux driver for local access and as an NVMe-oF initiator. For NVMe-oF, we used SPDK as the target-side driver. InfiniBand RDMA was used for transport.



(b) Our distributed driver is used for operating local and remote devices. We connected two hosts using PCIe adapter cards and a Dolphin cluster NTB switch to access the remote NVMe device over PCIe.

Fig. 9: The scenarios used in our latency benchmark evaluation: stock Linux drivers for a local device and NVMe-oF initiator were compared to using our driver implementation.

performance benefit of our solution lies. We have performed a synthetic random read/write benchmark using the Flexible I/O Tester (FIO) [17], version 3.28. Tests ran for 60 seconds, using 4 kB read/write size, with a queue depth of 1 to evaluate the network latency rather than disk performance. All tests were run on Ubuntu 22.04 (kernel 5.15.0-122). We used an Intel Optane P4800X NVMe as our target disk to avoid caching effects, as its latency is very consistent.

Figure 9 shows the two scenarios used in our experiment:

- **Using Linux drivers** (Figure 9a): we ran the read/write benchmark on a local host using the stock Linux NVMe driver to get a local baseline for the NVMe-oF comparison. A second host was connected using NVIDIA ConnectX-5 network cards and running OFED version 24.07-0.6.1.0. We configured NVMe-oF using RDMA as transport. As the initiator, we used the stock kernel's NVMe driver. On the target side, we used Stor-

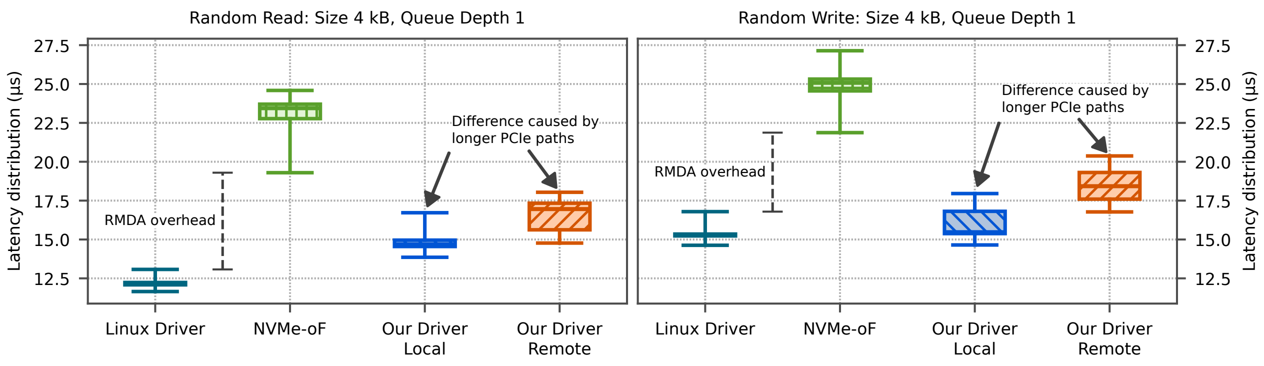


Fig. 10: I/O command completion latency for our benchmarking scenarios. Since our driver is not as mature as the kernel's NVMe driver, it has a higher baseline latency. However, our driver has less network overhead than NVMe-oF using RDMA, only adding the difference from the increased distance between the CPU and the NVMe device in the PCIe path.

age Performance Development Kit (SPDK) [18] version `g09cc66129` as the driver to minimize latency and provide the best performance. It should also be mentioned that we also attempted target offloading, but this only appeared to reduce CPU usage and did not affect latency.

- **Using our driver** (Figure 9b): similar to the Linux drivers scenario above, we first used our own driver for a local NVMe device to get a local baseline. Then, we connected a second host using Dolphin MXH932 adapters and an MXS924 NTB cluster switch and repeated the experiment for a remote NVMe. In the remote scenario, the NVMe device is farther from the CPU using it: each PCIe switch chip in the path adds between 100 and 150 nanoseconds delay (in one direction) for each PCIe transaction [5], [10].

Note that while our evaluation focuses on the difference between accessing a local and remote NVMe device using only two hosts, the sharing allows multiple hosts to use the device simultaneously. The P4800X used in our experiments supports up to 32 queue pairs (where one pair is reserved for the admin queues), and we have confirmed that it can be shared by up to 31 hosts simultaneously using our previous user-space implementation.

Figure 10 depicts the latency measurements for all four tests as boxplots. The whiskers depict the range from the minimum to the 99th percentile. Compared to the stock Linux driver, our driver implementation is naive. For example, we rely on polling instead of using interrupts. Additionally, as mentioned in Section V, our driver uses a bounce buffer as a workaround for NTB mapping constraints and, consequently, needs to do an additional memory copy in the critical path. However, the intention of our NVMe driver is not to outperform the Linux NVMe driver. Rather, the goal is to utilize NTB capabilities to enable low-latency sharing of NVMe devices. NVMe-oF has a significant network latency overhead compared to PCIe. The difference in *minimum* read latency is 7.7 µs for NVMe-oF vs. local, while it is around 1 µs for our implementation. For write, the difference in the minimum latency is 7.5 µs for NVMe-oF vs. local and around 2 µs for our implementation. The very low latency for remote in the tests of our own implementation corresponds with the increased distance between CPU and device and more PCIe switch chips in the path. While being a naive implementation and having some software overhead in general, we argue that our implementation does not add any *additional* overhead when operating a remote device beyond what is expected from native PCIe.

VII. Related Work

With our extension to the SISCI API, we have extended our previous work and implemented a working proof-of-concept kernel space NVMe driver, as described in Section V. Several PCIe-based solutions disaggregate devices at the function level, allowing the distribution of individual (virtual) functions to multiple hosts [13], [10], [19], [20]. Multi-Root I/O Virtualization (MR-IOV) [21] was an early attempt to allow PCIe devices to operate in multiple PCIe fabrics simultaneously, but it never saw any widespread adoption. More recent solutions are based on virtualization capabilities built into PCIe switch chips, such as those from Microsemi [20] and Broadcom [19]. These solutions allow stand-alone devices and computer hosts to be attached to a shared PCIe fabric. The switches' capabilities allow the PCIe fabric to be logically partitioned between the CPUs so that individual device functions can become dynamically hot-added or hot-removed while all systems are running. However, due to the complexity of implementing multiple device functions (either physical or virtual) in hardware, few NVMe devices on the market support this.

To the best of our knowledge, our driver is unique as it distributes individual I/O queues of a single-function NVMe device to *remote* systems in a cluster without using RDMA. By allowing multiple remote hosts to operate a single NVMe controller simultaneously, our distributed driver is, in practice, "software-enabled MR-IOV". Similar ideas for sharing an NVMe device at the queue level for user-space applications running on the same *local* system can be found in several implementations, including SPDK [18]. Peng et al. [22] implements a para-virtualized NVMe driver for assigning individual I/O queues to virtual machines. Similarly to our implementation, the authors rely on polling instead of device-generated interrupts for CQ interrupts. Kim et al. [23] 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 user-space application. This way, the 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. This solution is conceptually very similar to how our driver is implemented, but we support assigning queues to *remote* hosts as well. Supporting this kind of flexibility while allowing software to remain agnostic about the underlying PCIe topology is, to the best of our knowledge, a novel contribution.

VIII. Conclusion

In this paper, we have presented our NVMe driver implementation for sharing single-function NVMe devices among remote hosts in a PCIe cluster without using RDMA. By using shared memory segments to distribute I/O queues, we allow hosts to operate a remote NVMe in parallel. This could be useful for embedded systems where power and cost constraints make multiple drives infeasible.

Moreover, our experimental results show that by enabling direct access over PCIe, there is a significant latency benfit. However, the performance evaluation also showed that our naive driver lacks the same performance optimizations as the kernel NVMe driver. Therefore, a candidate for future work is to perform a thorough evaluation of the implementation and investigate performance bottlenecks and possible optimizations. Additionally, performing experiments using our driver for more general use, such as measuring performance when using a file system and realistic workloads, would contribute to validating our solution.

Finally, the upcoming Compute Express Link (CXL) standard could provide new opportunities and should be explored once CXL implementations become more widely available. Particularly, CXL 3.1 is interesting in the context of memory disaggregation and device sharing, and many concepts are similar to those of our SmartIO abstraction layer. For example, byte-addressable storage devices are of interest with regard to the memory-mapping capabilities of the SISCI API, which our driver implementation builds on.

References

[1] *NVM Express Base Specification*, https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3d-2019.03.20-Ratified.pdf, NVM Express, Mar. 2019, revision 1.3d.

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

[3] A. Taherkordi, F. Zahid, Y. Verginadis, and G. Horn, "Future Cloud System Designs: Challenges and Research Directions," *IEEE Access*, vol. 6, 2018.

[4] *NVM Express over Fabrics*, https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf, NVM Express, Oct. 2019, revision 1.1.

[5] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. K. Stensland, and C. Griwodz, "SmartIO: Zero-overhead device sharing through PCIe networking," *ACM Transactions on Computer Systems*, vol. 38, no. 1–2, jul 2021.

[6] Dolphin Interconnect Solutions, "SISCI API Documentation," http://ww.dolphinics.no/download/SISCI_DOC_V2/, 1999, accessed: 2024-09-24.

[7] *Single-root I/O Virtualization and Sharing Specification*, Peripheral Component Interconnect Special Interest Group (PCI-SIG), 2010.

[8] Z. Guz, H. Li, A. Shayesteh, and V. Balakrishnan, "NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation," in *Proceedings of the International Systems and Storage Conference*, ser. SYSTOR, May 2017, pp. 1–9.

[9] A. Daglis, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot, "Many-core Network Interfaces for In-Memory Rack-Scale Computing," *ACM SIGARCH Computer Arch. News*, vol. 43, no. 3, pp. 567–579, 2015.

[10] J. Markussen, L. B. Kristiansen, H. Borgli, H. K. Stensland, F. Seifert, M. Riegler, C. Griwodz, and P. Halvorsen, "Flexible Device Compositions and Dynamic Resource Sharing in PCIe interconnected Clusters using Device Lending," *Cluster Computing*, vol. 23, pp. 1211–1234, 2020.

[11] V. Meduri, "A Case for PCI Express as a High-Performance Cluster Interconnect," https://www.hpcwire.com/2011/01/24/a_case_for_pci_express_as_a_high-performance_cluster_interconnect/, 2011.

[12] X. Yu, "NTB: Add support for AMD PCI-Express Non-Transparent Bridge," https://lwn.net/Articles/672752/, Jan. 2016.

[13] PLX Technology, "Multi-Host System and Intelligent I/O Design with PCI Express," 2005.

[14] M. Ravindran, "Extending Cabled PCI Express to Connect Devices with Independent PCI Domains," in *Proceedings of the IEEE Systems Conference*, ser. SysCon, 2008, pp. 1–7.

[15] S.-G. Kim, Y.-W. Lee, S.-H. Lim, and K.-H. Cha, "Switchless Interconnect Network with PCIe Non-Transparent Bridge Interface," in *Advances in Computer Science and Ubiquitous Computing*. Singapore: Springer Singapore, 2020, pp. 97–102.

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

[17] J. Axboe, "Flexible I/O Tester," https://github.com/axboe/fio, 2005, accessed: 2024-07-03.

[18] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A Development Kit to Build High Performance Storage Applications," in *Proceedings of International Conference on Cloud Computing Technology and Science*, ser. CloudCom, 2017, pp. 154–161.

[19] I.-H. Chung, B. Abali, and P. Crumley, "Towards a Composable Computer System," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia, Jan. 2018, pp. 137–147.

[20] V. Haché, "Multi-Host Sharing of NVMe Drives and GPUs Using PCIe Fabrics," http://www.symmttm.com/document-portal/doc_download/1244483-multi-host-sharing-of-nvme-drives-and-gpus-using-pcie, Oct. 2019, accessed: 2024-09-24.

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

[22] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC, Jul. 2018, pp. 665–676.

[23] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs," in *Workshop on Hot Topics in Storage and File Systems*, ser. USENIX HotStorage, Jun. 2016, pp. 41–45.