Device Lending in PCI Express Networks

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

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

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

ABSTRACT

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

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

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

CCS Concepts

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

Keywords

Multimedia, GPU, PCIe, interconnect, device sharing

1. INTRODUCTION

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

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

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

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

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

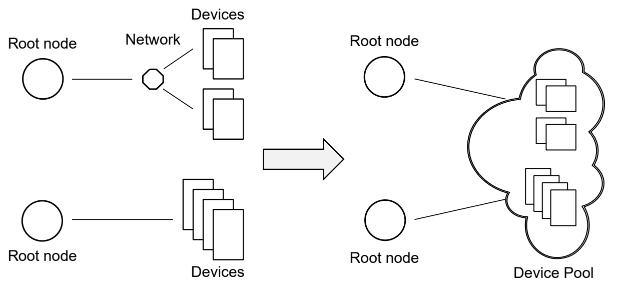


Figure 1: PCIe devices on separate machines could be pooled together and shared between multiple computers.

NOSSDAV'16, May 13 2016, Klagenfurt, Austria

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

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

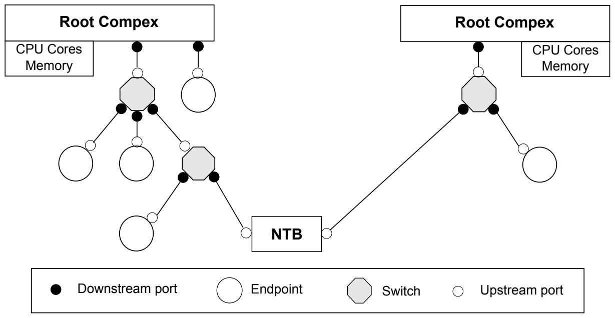


Figure 2: An example of a PCIe topology.

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

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

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

2. PCI EXPRESS

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

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

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

2.1 Memory-mapped IO

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

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

2.1.1 Posted and non-posted transactions

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

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

2.1.2 Transparent bridges

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

2.1.3 Non-transparent bridges

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

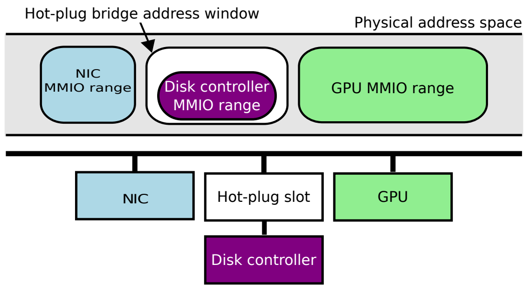


Figure 3: Physical address ranges are reserved by OS or BIOS at boot time. Memory requirements of hot-plugged devices must fit within the already existing address windows.

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

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

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

2.2 Message-signaled interrupts

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

2.3 Hot-plugging

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

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

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

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

3. VIRTUALIZATION SUPPORT IN PCIE

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

3.1 IO Memory Management Unit

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

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

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

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

3.2 Single-Root IO Virtualization

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

3.3 Performance penalty

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

4. RELATED WORK

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

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

4.1 Alternative protocols

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

4.2 Multi-Root IO Virtualization

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

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

4.3 Ladon and Marlin

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

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

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

5. IMPLEMENTATION

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

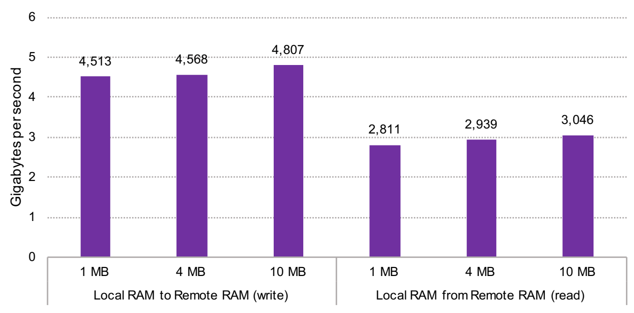
The lending side kernel module binds itself as a driver for the targeted PCIe devices. This provides us with exclusive access to the device, allowing the kernel module to access the device's configuration space while preventing other drivers on the host from interfering. The kernel module then notifies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

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

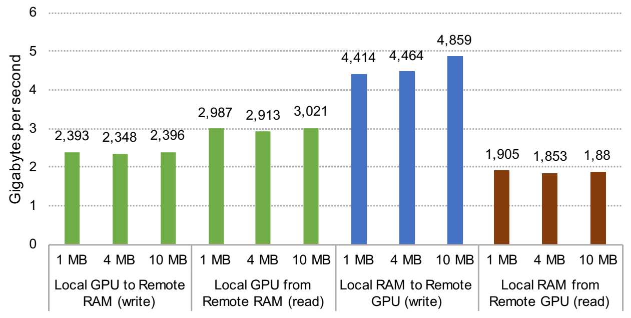
The device driver is now able to communicate with the device using MMIO access. Whenever the device driver sets up new DMA mappings using the Linux DMA-API, the borrowing side kernel module intercepts these calls and dynamically sets up and tear down the necessary IOMMU mappings. This allows the borrowing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

6. EVALUATION AND DISCUSSION

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



(a) RAM to RAM



(b) RAM to GPU

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

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

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

6.1 Reference evaluation

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

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

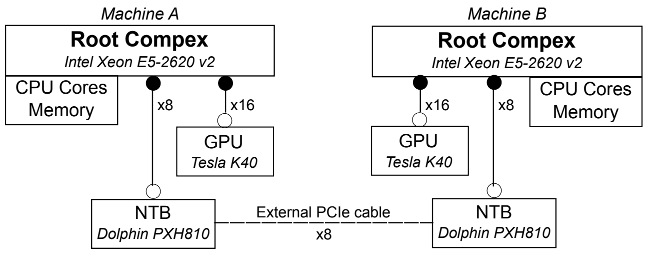


Figure 5: The setup used for our evaluation

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

6.2 Device Lending evaluation

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

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

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

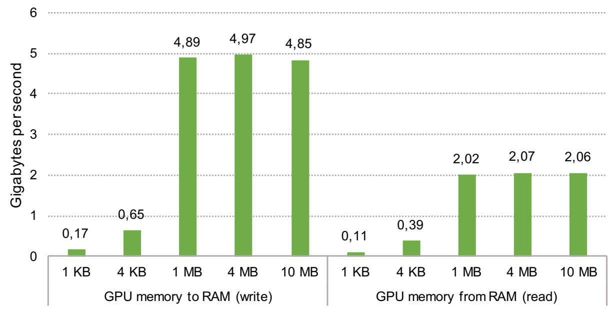


Figure 6: `bandwidthTest` running on a borrowed GPU. The DMA engine on the GPU is used to transfer.

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

7. CONCLUSION AND FUTURE WORK

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

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

Acknowledgments

This work has been performed mainly in the context of the BIA project *PCIe* (#235530) funded by the Research Council of Norway (RCN), with contributions from EONS (RCN #231687) and POPART (EU H2020 #644874). The authors also acknowledge Magma for providing Nvidia Tesla GPUs.

8. REFERENCES

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

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

[3] Dolphin Interconnect Solutions. SISCI API.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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