Understanding Routable PCIe Performance for Composable Infrastructures

Wentao Hou[1], Jie Zhang[2], Zeke Wang[2], and Ming Liu[1]

[1]University of Wisconsin-Madison [2]Zhejiang University

Abstract

Routable PCIe has become the predominant cluster interconnect to build emerging composable infrastructures. Empowered by PCIe non-transparent bridge devices, PCIe transactions can traverse multiple switching domains, enabling a server to elastically integrate a number of remote PCIe devices as local ones. However, it is unclear how to move data or perform communication efficiently over the routable PCIe fabric without understanding its capabilities and limitations.

This paper presents the design and implementation of **rPCIeBench**[1], a software-hardware co-designed benchmarking framework to systematically characterize the routable PCIe fabric. rPCIeBench provides flexible data communication primitives, exposes end-to-end PCIe transaction observability, and enables reconfigurable experiment deployment. Using rPCIeBench, we first analyze the communication characteristics of a routable PCIe path, quantify its performance tax, and compare it with the local PCIe link. We then use it to dissect in-fabric traffic orchestration behaviors and draw three interesting findings: approximate max-min bandwidth partition, fast end-to-end bandwidth synchronization, and interference-free among orthogonal data paths. Finally, we encode gathered characterization insights as traffic orchestration rules and develop an *edge constraints relaxing* algorithm to estimate PCIe flow transmission performance over a shared fabric. We validate its accuracy and demonstrate its potential to provide an optimization guide to design efficient flow schedulers.

1 Introduction

Composable infrastructures–organizing computing, memory, and storage as elastic resource pools–have gained a rising attraction recently. Empowering by emerging cluster interconnects [7, 8, 16], applications running over such a platform can access disaggregated hardware resources natively as local ones, adaptively scale based on workload demands, and achieve fine-grained sharing with co-located tenants, yielding independent scaling capability, high device utilization, and cost-efficiency improvement. We have seen a number of early engineering samples and commodity prototypes, such as GigaIO's FabreX [10], Liqid's SmartStack [13], H3's Falcon [12], Groq's GroqRack [11], and Enfabria's ACF [9].

PCIe (Peripheral Component Interconnect Express) is the defacto interconnect for high-performance intra-host communications. With the introduction of a specialized non-transparent bridge (NTB) device, one can extend the PCIe bus tree and facilitate communications between PCIe devices from different switching domains, enabling inter-host PCIe transactions or *routable PCIe*. Based on this capability, we can interconnect tens to hundreds of PCIe devices using NTB-enabled PCIe switches and adapters, which lays the foundation for many of today's composable infrastructures. More importantly, routable PCIe also serves as the basis for emerging memory fabrics, like CXL [8].

However, our community lacks a systematic understanding of the capabilities and limitations of routable PCIe. Specifically, first, as a routable PCIe fabric introduces extra external hops to PCIe transitions, what are the latency and bandwidth overheads? Second, since the fabric concatenates another PCIe switching domain at the endpoint of the local PCIe bus tree, compared with the intra-host PCIe link, how does the inter-host link behave? How well does it orchestrate co-located flows? Third, the routable PCIe fabric allows concurrent host-device and device-device communications. Since the existing PCIe layered protocol still applies with no changes, when different communication paths interleave, how do they interact with each other? In sum, there is a strong need to characterize the routable PCIe fabric, firmly answer these questions, and derive some design guidelines to assist in building communication sublayers and runtime systems atop routable PCIe-enabled composable infrastructures.

Toward this end, we design and implement a software-hardware co-designed benchmarking framework (called **rPCIeBench**) to help us conduct the characterization study. It consists of three major components: (1) programming APIs that provide various data movement primitives and allow developers to configure arbitrary testing scenarios; (2) host run-

[1]rPCIeBench is available at https://github.com/netlab-wisconsin/rPCIeBench.

time and driver, responsible for both data-plane PCIe transaction delivery as well as control-plane platform management and profiling; (3) FPGA bitstream, realizing the device-side logic and manifesting itself as a reconfigurable target accelerator. Overall, rPCIeBench is generic and device/fabric-independent, enables end-to-end PCIe transaction observability, and allows flexible HW/SW/traffic configurations.

We apply rPCIeBench to GigaIO's FabreX testbed [10] and first examine the performance characteristics of one routable PCIe path. We find routable PCIe indeed incurs performance tax. Its one-way PCIe latency between two endpoints is 868.6 ns, while a local one takes 379.0 ns. The forwarding rate of an external PCIe switch is slower than a server internal one, yielding 30.4% and 6.9% bandwidth degradation for host→device and device→host scenarios. Further, as routable PCIe hinges on the credit-based flow control, the more intermediate hops along a routable PCIe path, the more time it takes to replenish credits, resulting in higher latencies, especially when bandwidth is oversubscribed. Our findings also indicate that an external PCIe link preserves most proprieties of a local one due to the inherent same layered protocol architecture. For example, the bandwidth partition among concurrent PCIe flows over a link depends on the ratio of their outstanding bytes. PCIe is bidirectional, imposing little interference between concurrent reversed flows, regardless of local or external.

We then use rPCIeBench to dissect in-fabric traffic orchestration characteristics and draw three findings. First, in a routable PCIe fabric, each communication port realizes a *credit-by-credit* round-robin scheduling discipline across active lanes, yielding an approximate max-min bandwidth partition. Second, the fabric preserves little buffering at adapters and switches, where the bandwidth availability can be piggybacked via credits and quickly back-propagated from the congestion point to other parts along the path. Third, orthogonal data communication paths over the routable PCIe fabric can be viewed as physically isolated communication domains, imposing little performance interference.

Finally, we formalize the data movement problem over a routable PCIe fabric, encode our empirical findings as traffic orchestration rules, and derive a solution to estimate flow transmission performance. Our *edge constraints relaxing* algorithm takes the underlying fabric topology and PCIe flow properties as inputs, applies iterative reduction by gradually constraining flow bandwidth based on the capacity of oversubscribed links, and outputs the per-flow achieved bandwidth. Our characterization insights make the routable PCIe fabric well-structured and predictable, holding great potential to assist flow scheduling design. We validate the algorithm in three different experimental settings and show that the average performance prediction error rate is 2.9–11.3%.

2 Background

This section provides the necessary background about routable PCIe and the resulting composable infrastructures.

2.1 PCIe Non-Transparent Bridge and Routable PCIe

PCIe [16], introduced in 2003, is an interconnect for communication among processors and peripheral devices. It is a packet-based data communication network and provides point-to-point connections through high-speed serial buses. PCIe is organized into three layers: (a) physical layer, which transmits/accepts packets over a link and performs packet encoding/decoding; (b) data link layer, maintaining data integrity, sequencing packets from the transaction layer, and ensuring reliable delivery via the credit-based flow control protocol [38–40]; (c) transaction layer that realizes different request and completion transaction semantics. Today, PCIe Gen3/4 devices and ecosystems are predominant, Gen5/6 is gaining adoption, and industry standardization of Gen7 is underway and expected to be finalized in 2025.

Generally, a PCIe interconnect network consists of endpoints, switches, bridges, and root complexes, running under one memory domain within a host and supporting the corresponding layer functionalities. A bridge, switch, and root complex forwards and routes packets using memory-mapped I/O (MMIO) addresses or requester IDs. To enable cross-host PCIe communication, a special type of PCIe bridge device–PCIe Non-Transparent Bridge (NTB)–is introduced. A PCIe NTB allows a local host to interact with a remote device via native PCIe transactions by building two memory address mappings: (1) between a remote host and a local NTB: and (2) between an NTB and a local host. As such, one can enable *routable PCIe* traversing through multiple hosts without sharing the same memory domain. To realize scalable deployment, one can integrate a PCIe NTB into an external PCIe switch that interconnects tens of remote PCIe devices. Consequently, these remote PCIe devices will appear in the host PCIe subsystem as a PCIe subtree, laying out the foundation for composable infrastructures. More importantly, routable PCIe has become the basis of emerging memory fabrics (such as CXL [8] and CCIX [7]).

2.2 Composable Infrastructures

Infrastructure composability has gained significant attraction recently because of its independent scaling capability, high device utilization, and improved cost efficiency. By exposing remote accelerators and I/O devices as local, applications can access a large pool of computation/storage resources using native PCIe or other interconnect transactions (without protocol conversion), adaptively scale based on workload demands, and achieve fine-grained sharing with co-located tenants. We have seen a rising number of infrastructure startups delivering a variety of solutions, such as GigaIO's FabreX [10], Liqid's SmartStack [13], H3's Falcon [12], Groq's GroqRack [11], and more. We use the FabreX system as the developing target, and our benchmarking system generally applies to others. Figure 1-a and -b depict our prototyped composable testbed and the architecture of a typical routable PCIe fabric. It encloses (1) a couple of external PCIe switches that realize scalable

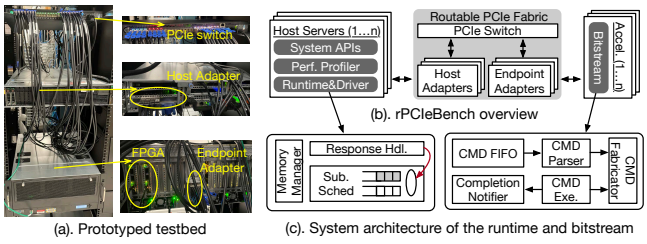


Figure 1: The software architecture of rPCIeBench and its targeted composable hardware testbed.

topologies; (2) host PCIe adapters, offering server-side connectivity; (3) endpoint PCIe adapters, which hold accelerators and I/O devices in standalone chassis. All connections use PCIe copper SFF-8644 cables [18]. There is a fabric manager deployed at one dedicated server, responsible for system management, such as device enumeration, topology configuration, and liveness monitoring.

As discussed above, routable PCIe is the technology enabler to build composable infrastructure. However, our community lacks a systematic understanding and detailed performance characterization of routable PCIe, especially when communicating with composable devices. There are no canonical software utilities, test suites, or referenced hardware platforms. Thus, we fill this gap by developing a benchmarking framework (called **rPCIeBench**). Based on it, we design various experimental composable scenarios, dissect how routable PCIe interacts with remote devices, and analyze its in-fabric traffic characteristics.

3 rPCIeBench Framework

This section first describes the design principles of rPCIeBench, and presents its system design and implementation.

3.1 Design Principles

Our goal is to systematically characterize the performance of routable PCIe and analyze its execution behavior under different composable scenarios. We build rPCIeBench adhering to the following principles:

- **Generality.** rPCIeBench supports any PCIe-based hosts and routable PCIe fabric testbeds, not relying on device-dependent functionalities. We divide the benchmarking functionalities between host servers and target devices;
- **End-to-end operation.** rPCIeBench should capture the communication performance of an entire data movement between the data generator and data consumer. We enable end-to-end tracing and equip a reconfigurable accelerator at the target side to interact with hosts flexibly;
- **Reconfigurability.** rPCIeBench should be able to generate stipulated benchmarking requests based on a traffic profile. We expose a set of programmable APIs, open-source the reference hardware architecture and software implementation, and define pluggable interfaces for module updates.

3.2 Overview

rPCIeBench consists of three components (Figure 1-b), spreading across host servers and remote devices. The first one is programming APIs that allow developers to implement and deploy arbitrary testing scenarios. Users prescribe benchmarking servers and target devices, initialize the system environment, and configure data movement patterns and attributes. The second part is the host runtime and driver, responsible for fabricating and submitting PCIe requests, interacting with the underlying PCIe subsystem and host adapter, handling transaction completions, and conducting performance analyses. The last piece is the bitstream within the FPGA accelerator. An FPGA generally encloses programmable LUTs (lookup tables), DSPs (digital signal processors), domain-specific engines, and heterogeneous memory domains (like block RAMs and HBMs), enabling us to emulate different types of data communications. Specifically, our bitstream sets up the FPGA execution environment, receives data transfer requests, instantiates a series of data transfers via DMA engines over routable PCIe fabric, reads/writes data to memory destinations, and issues completion signals back.

3.3 System APIs

rPCIeBench provides three types of APIs. The first one is used to initialize the execution environment of remote FPGAs, configure the device memory, and set up the host-device address mapping. The second category allows device-side memory management such that one can specify the source and destination of memory locations for a data transfer. The last one offers generic communication primitives, enabling host-device and device-device data movement via the MMIO (memory-mapped I/O) or DMA engine. We equip each primitive with several attributes, such as performing batched communications via a scatter-gather list, enabling flexible load balancing among multiple queues, and more.

3.4 Software Components

rPCIeBench benchmarks and characterizes the routable PCIe fabric using three software subsystems (Figure 1-b).

Performance Profiler. We trace a PCIe transaction's entire lifetime, from when the benchmarking application submits the requests until receiving the completion signals. Our utility timestamps the transaction queueing time at the host server (phase 1), data traversing time over the fabric (phase 2), and command execution at the remote accelerator (phase 3). All timestamps are marked at the nanosecond precision. We use polling to improve the system profiling accuracy. After all stipulated requests are finished, we report (a) the overall bandwidth, queueing, and average/tail latency; (b) the CDF/histogram of each transaction and its individual phases. We follow design strategies (e.g., bitwise recording format, compact data structure, and memory logger) of contemporary perf tools [15, 17] when building the profiler.

Category	API	Description
Device Conf.	`dev_init (pci_bus_addr, bar_addr, size)`	Initialize the FPGA device and map the corresponding BAR address
	`dev_mem_init(pci_bus_addr, region)`	Instantiate the memory manager for the given FPGA memory region
	`dev_setup_mapping(pci_bus_addr, region, hmem_addr)`	Map the FPGA's region to the host and set up the device mapping
Memory Mgt.	`dev_mem_alloc (pci_bus_addr, region, size)`	Allocate the device memory from a given region of an FPGA
	`dev_mem_free (pci_bus_addr, dmem_addr)`	Free the allocated address from an FPGA and clear up the mapping
	`dev_mem_getaddr (pci_bus_addr, hmem|dmem_addr)`	Obtain the memory-mapped host(device) address
Communication	`mmio_rd|wr(pci_bus_addr, hmem_addr, dmem_addr, size)`	Perform an MMIO read/write from the host to a device
	`h2d|d2h(pci_bus_addr, hmem_addr, size, dmem_addr, qnum)`	Move data between the host and the device via a given DMA queue
	`dev2dev_rd|wr(pci_bus_addr, dmem1_addr, size, dmem2_addr, qnum)`	Move data between two FPGA devices via a given DMA queue

Table 1: The rPCIeBench API list. hmem/dmem = Host(Device) memory address. All communication APIs support a batched version.

Runtime & Driver. Our system runtime has three parts: (a) a memory manager that allocates and reclaims host-side memory for data movements; (b) a request submission scheduler, determining the next issuing transaction based on the specified policy; (c) a response handler, which polls the completion vector and wakes up the corresponding submission path. We use the Linux *hugepage* and implement a segment-based memory allocator [23, 24] atop. As shown in Figure 1-c, the scheduler is a multi-queueing system, exposing a programmable interface for users to limit the number of outstanding requests and define the scheduling policy. One can further control the scheduling behavior at a fine granularity for each queue. Besides, our driver layer realizes a slim PCIe subsystem that implements the basic functionalities (such as bus enumeration, device registration, and buffer/engine management) to interact with the device on the control plane (using memory-mapped registers) and data plane (through DMA).

Bitstream. Figure 1-c depicts the circuit diagram of the remote accelerator. It has three 64-bit base address register (BAR) spaces for different roles. BAR0 is used for configuring the DMA engine, and BAR2 enables passing benchmarking parameters. BAR4 is connected with the FPGA's HBM and mapped to the host memory for data movement. One can also use BlockRAM in this case and we present the latency comparison in Appendix B. There are five modules along the command execution pipeline: (a) command FIFO queues, taking user requests via MMIO write, where the host runtime specifies the queue ID; (b) command parser, analyzing the request format, extracting the parameters, and checking the request's validity; (c) command fabricator, which encapsulates PCIe transactions and submits them to the DMA engine; (d) command executor, reading from device-side memory, buffering data temporarily, and issuing PCIe writes to the host memory under host→device communications (device→host works vice versa); (e) completion notifier, writing the completion signal to a predefined memory region. Note that (1) host→device and device→host, albeit exhibiting similar processing paths, use different hardware components; (2) we realize device↔device communications by mapping one FPGA's HBM to the host memory and accessing it via another FPGA's DMA engine, causing data copied from one FPGA to another.

3.5 Command Data Path

rPCIeBench supports three types of communication primitives (Table 1). An MMIO read/write, issued from the host processor, is the first category, generating only one PCIe read/write transaction to access the device memory. The second one is a host-device data movement. As depicted in Figure 2-a/b, it encompasses four steps: (a) passing command arguments via an MMIO write, (b) moving data between host and device, (c) reading/writing to the HBM, and (d) issuing completion signals via another PCIe write, yielding 1 MMIO write and 2 PCIe transfers (which will translate to multiple PCIe transactions based on the command size) in total. The last type is device-device communication (Figure 2-c/d), operating similarly to the host-device case. The difference is that two device memory accesses are triggered at both source and destination. We use a server host to submit requests and catch completion signals. In our implementation, command FIFO queues and data buffer (of the command execution engine) are located in the block RAMs (BRAMs). Under batch execution, the command fabricator within each device (Figure 1-c) formalizes a list of transactions and schedules them concurrently. We trace each primitive between submitting the command and receiving the completion acknowledgment.

3.6 Workflow

Using the rPCIeBench framework requires three basic steps, and we follow them when performing the characterization throughout this paper. First, one should configure the composable testbed based on the experimental data movement flows, considering how host adapters, PCIe switches, and endpoint adapters are connected. The second step is to write profiling applications using our system APIs. This includes determining traffic profiles and benchmarking parameters. Finally, one will deploy the host execution environment, load the bitstream into FPGAs, run the profiling application, and collect performance results.

4 Basic Performance of Routable PCIe

This section examines the performance characteristics of routable PCIe and compares them with the local PCIe case.

4.1 Experimental Methodology

Hardware testbed. Our host servers are 2U Dell R740 boxes, enclosing two 20-core Intel Xeon Gold 6248 processors (running at 2.5GHz), 192GB DRAM, and 1.92TB HDD. We disable both Hyper-Threading and Turbo Boost features. All PCIe lanes of the server are Gen3. We use Xilinx Alveo U55C cards (×16) as the major fabric-attached devices. As discussed above, we choose the GigaIO's Fabrex as the

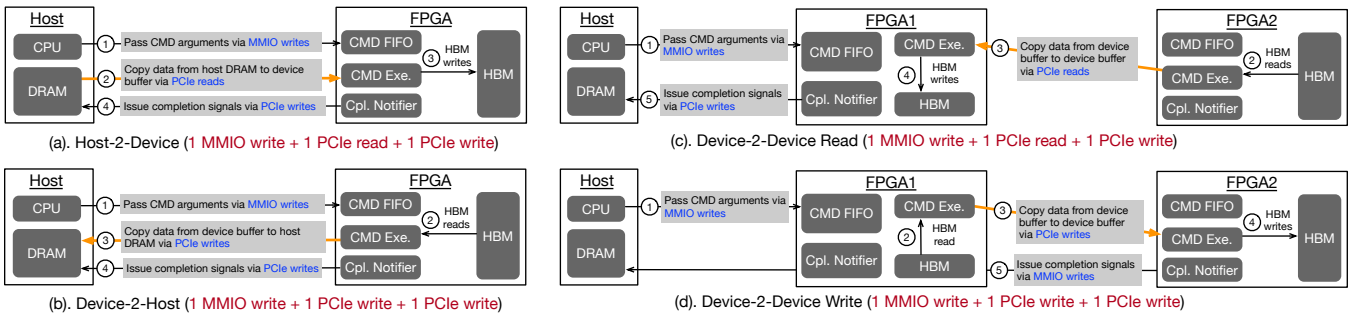
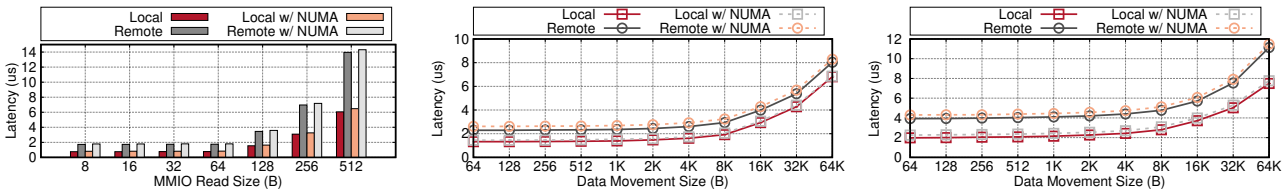


Figure 2: Data path of four communication primitives. We consider the data movement between host DRAM and device HBM.



(a) MMIO. **(b) Device→Host.** **(c) Host→Device.**

Figure 3: Latency of MMIO, Device→Host, and Host→Device communication when varying the data sizes.

routable PCIe-based composable testbed. Its RS4024 switch has 24 ports, where each connects to a PCIe Gen3×4 link.

Terminology. We use a PCIe flow to describe one data transfer from a source entity to a destination entity. Multiple flows can interleave over the same communication path for different data movements. A PCIe transaction layer packet (or packet for short) and a PCIe transaction are used interchangeably, referring to the smallest transmission granularity of a PCIe flow. Our work mainly considers three types of PCIe transactions [16]: memory read, completion with data, and memory write. The first two are non-posted, requiring data responses, while the last one is a simple posted transaction. MPS (maximum payload size) and MRRS (maximum read request size) limit the size of corresponding packets, which are 1024B and 512B in our case.

Experiment configuration. This section focuses on the single communication path. There are three types of communication paths in a composable testbed: host→device (H2D), device→host (D2H), and device→device (D2D). We set up each of them and use the rPCIeBench's communication primitives for traffic generation. We change our traffic profile by varying the number of outstanding PCIe flows, the packet size per flow, and its burstness. rPCIeBench reports average/tail latency and throughput as the major performance metrics.

4.2 Latency

One-way PCIe. We first dissect the one-way PCIe latency between two entities using the rPCIeBench' tracing functionality. When communicating within a server, we find out that the local PCIe one-way latency is 379.0 ns, which matches the number reported in recent literature [1, 2, 56]. However, when traversing across the routable PCIe fabric, the one-way PCIe latency rises to 868.6 ns, adding 489.6 ns (129.2%) overheads! This is non-trivial for small-sized PCIe transfers. We further worked with the device vendor and performed a latency breakdown. We find that (1) the host adapter, switch, and target adapter consume ∼105ns each due to the NTB switching, respectively; (2) the propagation delay of the copper wire is around 5ns; (3) the RS4024 has a 10ns processing delay; (4) the host-side software takes ∼150ns.

DMA-induced PCIe. When PCIe transfers are triggered via DMA, we should include the DMA engine execution cost, including preparing the command, submitting it to the command queue, and catching the completion signal. We examine the hardware module within the accelerator and find out this overhead is around 418.0 ns regardless of local or remote. For example, a 64B PCIe write issued via the DMA engine would take 946.0 ns and 1421.4 ns to complete in the local and remote cases, respectively.

MMIO & H2D & D2H. The latency of an MMIO read depends on the number of generated cache lines. As shown in Figure 3-a, a local 64B PCIe read takes 766.0 ns, while the remote one consumes 1751.0 ns, because one PCIe round trip (two-way) is required. When crossing the CPU socket, we observe there is an additional 67.0 ns and 52.0 ns for the local and remote scenario, contributing 833.0 ns and 1803.0 ns, respectively. When the MMIO read size is 1KB, yielding 16 cache lines, a local PCIe latency rises to 11.9 us, while the remote one increases to 27.8 us.

Both device→host and host→device trigger the same amount of PCIe transactions (Figures 2-a/b). However, in the D2H case, as we overlap the data write and completion acknowledgment, it takes less time to finish. Figures 3-b/c report our results. For example, a 64B D2H data movement consumes 1.3 us, while the H2D takes 2.0 us. When the data size is less than 4KB, routable PCIe adds 69.3% and 91.5%

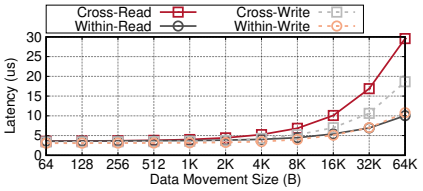


Figure 4: D2D latency when communication within or across a PCIe switch.

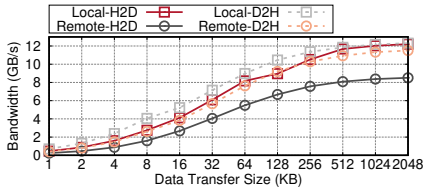


Figure 5: H2D&D2H bandwidth varying the data transfer size.

Figure 6: D2D bandwidth when communicating within or across a PCIe switch.

latency penalties to the D2H and H2D cases due to 3 one-way PCIe. With larger data movement sizes, such overheads diminish considerably. For instance, when performing a 64KB data transfer over the routable PCIe fabric, the D2H/H2D path takes 8.0/11.1 us, adding 17.9%/48.7% (6.8/7.5 us) compared to the local scenario. This emphasizes the importance of batching when building systems over the routable PCIe fabric. Further, NUMA still hurts latency a little bit. On average across all cases, it brings in 10.7%/7.0% overheads for the local H2D/D2H data transfers and adds 7.5%/11.2% more latencies for the remote ones.

D2D. We focus on two types of device-device communication: one is crossing the external PCIe switch; the other is within the PCIe subtree, not across the switch. Clearly, traversing the switch is not free. When the data transfer size is less than 1KB, as shown in Figure 4, crossing the switch incurs 2.2% and 11.0% more latencies for the read and write scenarios, respectively. As the data movement size increases beyond 1KB, we find that the overhead increases significantly. For example, a 64KB data transfer over D2D read/write within the subtree consumes 10.0/10.8 us, but takes 29.6/18.6 us when passing the switch, resulting in 194.4%/72.6% overheads.

Takeaways. Communicating over the routable PCIe fabric (via the switch and adapters) is not as performant as the local case. A one-way PCIe transfer takes 868.6 ns (compared with 379.0 ns in the local case). When using DMA engines for data movements, one should also consider the engine execution cost (which is 418.0 ns in our case). Large data transfer (beyond 4KB) can amortize the routable PCIe-induced latency overheads for H2D and D2H scenarios, suggesting the effectiveness of batching. However, for D2D communication, traversing the external PCIe switch is costly, especially for 4+KB data sizes. This indicates that when building D2D communication subsystems, one should consider not only their positions over the fabric, but also the data transfer granularity.

4.3 Bandwidth

H2D&D2H. We gradually increase the data transfer size and measure the communication bandwidth (Figure 5). Within a server host, H2D and D2H max out their bandwidth with at least 1MB data granularity, achieving 12.2 GB/s and 12.3 GB/s. However, when communicating across the routable PCIe fabric, H2D and D2H drops to 8.4 GB/s and 11.3 GB/s, contributing to 30.4% and 6.9% degradation. We carefully examine each communication entity across the path and find out that the maximum payload size (MPS) and the number of concurrent PCIe transactions are the same in both local and remote cases. This indicates the bandwidth drop mainly comes from the fact the PCIe transaction rate of the external switch is slightly lower than an internal PCIe switch on the server board. The H2D and D2H have different performance degradation because the adapters and switch of our composable testbed use different DMA engines for upstream and downstream links, respectively.

D2D. Next, we present the device→device communication bandwidth. As shown in Figure 6, within a PCIe subtree (not across the remote switch), a read/write D2D transfer achieves 9.8/8.0 GB/s. However, surprisingly, when traversing the switch, the maximum achieved bandwidth is only 2.3/4.3 GB/s! The 4.2/1.8× bandwidth degradation cannot be simply attributed to the additional switching hop across the path (§4.2). By dissecting the data path (Figure 2), we find out that another limitation–root complex contention, happening because all the PCIe transactions (including launching, preparing, and running the command) pass the root of a PCIe bus tree–throttles the number of concurrent cross-switch D2D transfers. However, in the within case, step 3 (Figure 2-c) and steps 3/5 (Figure 2-d) are executed locally within the subtree.

Takeaways. The forwarding rate of an external PCIe switch operates slower than a server internal PCIe switch, yielding 30.4% and 6.9% bandwidth degradation for H2D and D2H scenarios. Device-to-device communications not only traverse the external PCIe switch but might also cause root complex contention (when devices are located in a local-remote hybrid scenario), jeopardizing the maximum achieved bandwidth.

4.4 Latency v.s. Throughput

We examine the latency-throughput relation for each data movement direction. We gradually inject more background traffic (generated via large PCIe transactions) and measure the average latency of 64B PCIe requests. As shown in Figures 7 and 8, the latency starts to rise when approaching the maximum bandwidth because credit starvation happens, causing request stalls. However, we find that it takes more time for a routable PCIe fabric to replenish credits. For example, when achieving 80–90% of the maximum bandwidth, the local H2D and D2H experience 20.2% and 28.7% higher latencies, while the remote ones see 55.3% and 28.1%, respectively. Similarly, within a PCIe subtree, there is an 18.3%/3.9% higher latencies for the D2D read/write case, while the cross fabric case

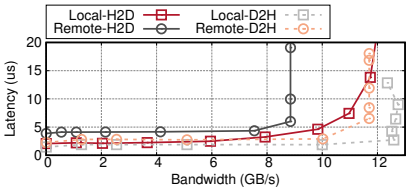


Figure 7: Latency v.s. throughput for local/remote H2D and D2H cases.

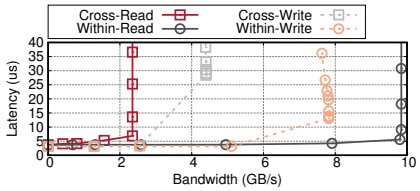


Figure 8: Latency v.s. throughput for within/cross D2D read/write cases.

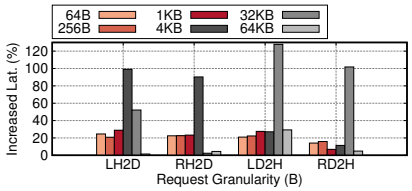
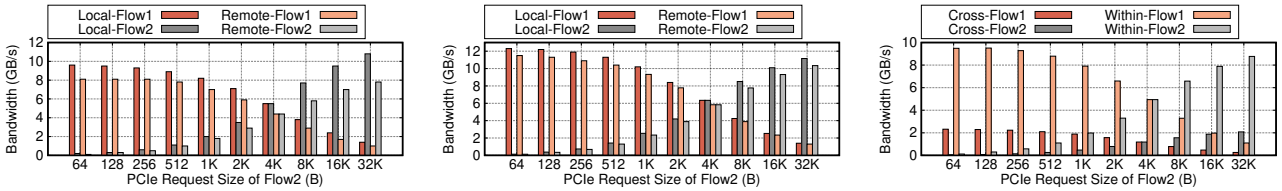


Figure 9: Tail-over-Avg latency increases for local/remote H2D and D2H cases.



(a) Host→Device. **(b) Device→Host.** **(c) Device→Device Read.**

Figure 10: Bandwidth partition between two concurrent flows of Host→Device, Device→Host, and Host→Device communications.

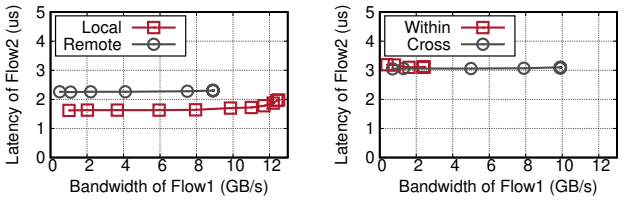
experiences 41.1%/6.7% more. Since the (routable) PCIe fabric applies a hop-by-hop credit-based flow control, the more intermediate entities along the path, the more credit interactions one would observe. When bandwidth is (close to) oversubscribed, a longer communication path needs more credit coordination to deliver a transaction.

Tail latency. Next, we interleave 16 concurrent homogeneous-sized PCIe flows and sweep the request size of each flow from 64B to 64KB. For each data movement direction, when the number of available credits runs out at the link layer, a PCIe transaction would be queued up, increasing the service latency. Hence, we measure the average/P99 latency and use the $\frac{Tail_{lat}}{Avg_{lat}}$ metric to estimate the credit capacity. As shown in the Figure 9, we find that the credit capacity is not consistent for different directions. For example, the H2D experiences the largest ratio under 16 4KB requests, generating up to 512 concurrent transactions, while the D2H direction can sustain 4096 ones (i.e., 16 32KB). This is the same for both local and remote cases, indicating that the routable PCIe fabric has provisioned enough credits (or communication resources) than endpoints. Similarly, D2D reads/writes support 128 and 512 concurrency when staying within and across the fabric, respectively.

Takeaways. Similar to most communication fabrics, one would experience latency rises under high bandwidth utilization. However, the issue stems from the credit-based flow control in the data link layer. It generally takes more time for a routable PCIe fabric to replenish credits because there are more intermediate identities along the path, requiring more credit coordination. The fabric is provisioned with more credits than endpoints, leaving itself from becoming a communication bottleneck from the data link layer perspective.

4.5 Bandwidth Partition

We explore how bandwidth is partitioned across concurrent PCIe flows. Our experiments are configured as follows. For each data movement direction, we consolidate two PCIe flows



(a) Device→Host. **(b) Device→Device.**

Figure 11: Path asymmetry demonstration of Device→Host and Device→Device communications.

that continuously issue one outstanding PCIe request: Flow1 sends a 4KB request; Flow2 increases its transaction size from 64B to 32KB. We find that when concurrent PCIe flows share the same communication path, bandwidth partition among these flows is roughly proportional to the ratio of their outstanding bytes. Take the H2D case as an example (Figure 10-a). When a 4KB flow contends with a 128B one, Flow1 and Flow2 achieve 9.5 GB/s and 0.39 GB/s, respectively, resulting in a 32.8 partition ratio. When two 4KB flows interleave, both sustain at 5.5 GB/s. The remote H2D scenario shows similar results. This observation also holds for the device→host data movement. For example, a 4KB flow achieves 4.3/3.9 GB/s in the local/remote D2H case (Figure 10-b), one-third of the total bandwidth, when intermixing with the 8KB flow. When moving data between two devices, such a bandwidth partition rule still holds. As shown in Figure 10-c, Flow1 only consumes 0.5 GB/s and 1.9 GB/s in the D2D read case when across or within the external PCIe switch, ∼22.0% of the total bandwidth, where Flow2 issues a 16KB request.

Takeaways. Between two endpoints, the bandwidth partition among concurrent PCIe flows mainly depends on the ratio of their outstanding bytes. The defacto transaction layer imposes no fair bandwidth enforcement. The routable PCIe fabric extends the basic scheme of a local PCIe network.

4.6 Asymmetric Communication Path

PCIe is a full-duplex bidirectional network. This section explores whether flows with opposite directions interfere with

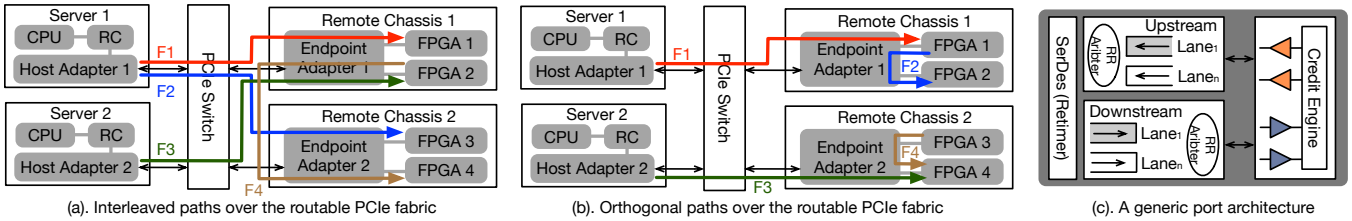


Figure 12: We consider a small deployment with two host servers and two remote chassis, connected via an external PCIe switch. (a) and (b) present in-fabric traffic sharing scenarios. (c) shows the generic architecture of a communication port. RC=Root Complex.

each other. We place a latency-sensitive flow (Flow2) from A to B and a throughput-oriented flow (Flow1) in the reverse direction B→A over one physical communication path, and then analyze how latency varies with the throughput. Figure 11 reports our results. Take the device→host as an example. When maxing out the bandwidth, Flow2's latency only increases from 1.6 us to 1.8 us in the local case, while the remote one stays around 2.2–2.3 us. Similarly, in terms of device-device communications, within a PCIe subtree, Flow2's latency sustains at 3.1 us regardless of how much traffic is injected on the reversed side; across the PCIe switch, Flow2's latency varies between 3.0 us and 3.1 us. Hence, there exists little interference among concurrent flows under opposite directions.

Takeaways. Akin to the local PCIe network, routable PCIe incurs no communication interference among concurrent reverse PCIe flows over one physical path, no matter whether transmitting data is in any of the following directions: host→device, device→host, and device→device.

5 In-Fabric Traffic Orchestration

§4 focuses on understanding different aspects of a single communication path. This section analyzes how multiple paths interact over the routable PCIe fabric, especially at the host adapter, external PCIe switch, and endpoint adapter.

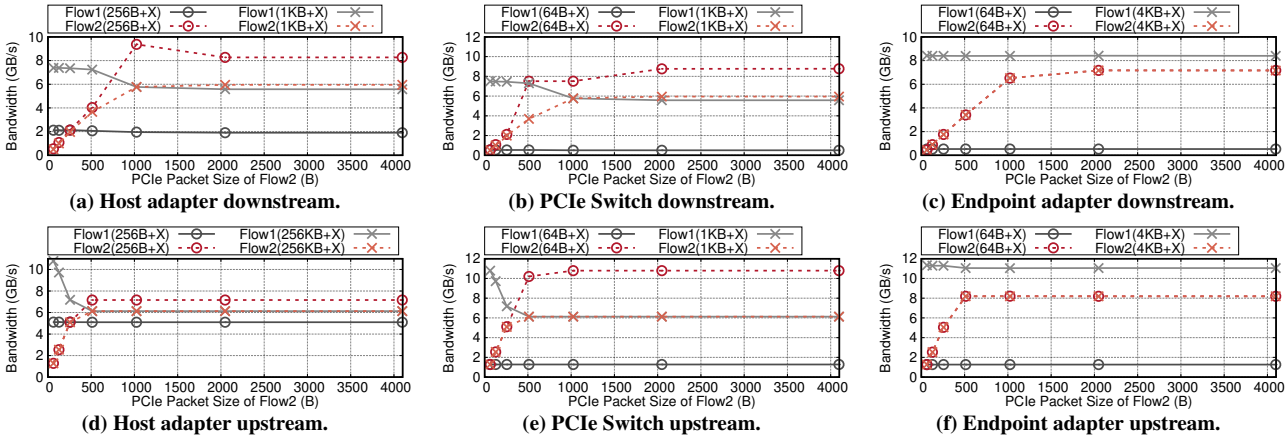
5.1 Max-Min Fair Bandwidth Allocation

Across the fabric, PCIe flows from different communication paths contend for the bandwidth resource of any intermediate transmit points. As shown in Figure 12-a, we configure three path interleaving scenarios that share the host adapter (F1 v.s. F2), switch (F1 v.s. F3), and endpoint adapter (F2 v.s. F4), respectively. In each experiment, we fix the packet size of one flow, gradually increase the packet size of another flow, and explore how bandwidth is partitioned.

Our results show that each communication entity (e.g., an adapter or a switch) realizes an approximate max-min bandwidth allocation scheme. Specifically, when N flows from different paths/lanes share an upstream/downstream port with the following demands $BW_{F_1}, BW_{F_2}, ..., BW_{F_n}$, if the aggregated bandwidth is less than the link capacity, each flow can achieve its desired rate; if the bandwidth is oversubscribed, each flow F_i will receive its max-min share.

For example, as shown in Figure 13-a, when a 256B flow is interleaved with a 4KB one at the downstream path of a host adapter, both max out their bandwidth, resulting in 10.7 GB/s, less than the link capacity. However, in terms of the 1KB and 4KB mixed case, they achieve 7.5 GB/s and 8.5 GB/s when running in a standalone mode, but receive an equal bandwidth share (i.e., 5.6 GB/s). The upstream one presents similar results (Figure 13-d). Regarding the PCIe switch (Figures 13-b/e), when a 64B flow (Flow1) shares with the other one, it can always achieve 0.5/1.3 GB/s along the downstream/upstream path. When Flow1's packet size rises to 1KB, Flow1 sustains at 7.5 GB/s if the packet size of Flow 2 is less than 512B, and drops to 5.6 GB/s, which is the same as Flow2 if the packet size exceeds 1KB. The endpoint adapter behaves similarly. Take the 4KB+X upstream contention as an example (Figures 13-f). Since two flows traverse different communications (one is host→device and the other is device→device), Flow1 and Flow2 achieve 12.6 GB/s and 8.2 GB/s at most if deployed exclusively. When interleaving, Flow2 is able to max out, but Flow1 is limited to 11.3 GB/s due to the link capacity. We also notice that the bandwidth partition at the upstream and downstream points is not always symmetric (Figure 13-c). We believe this is mainly due to the implementation differences across our communication primitives (e.g., the completion delivery step in Figure 2).

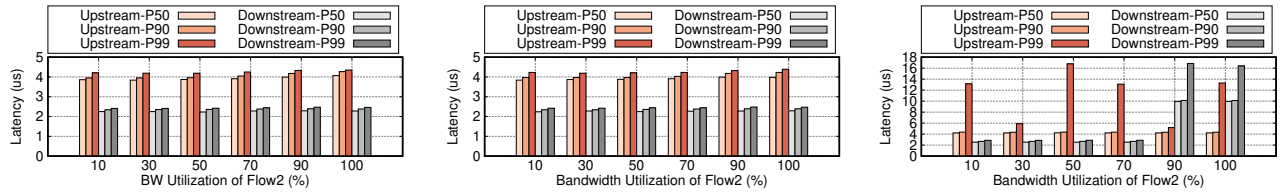
We then drill down to the underlying mechanism to explore how such cross-lane (link) max-min fairness is realized. By walking through the hardware details of the adapter/switch, we find that they all employ a generic port architecture (Figure 12-c), which includes: (a) a SerDes module for data conversion, (b) a upstream and downstream pipeline for packet transmission, and (c) a credit engine to realize the link layer protocol. Some might also include a PCIe retimer to retransmit signals. The reason why max-min fairness across lanes is guaranteed is due to the compounding effect between the credit engine and round-robin arbiter within the pipeline. Specifically, the credit-based flow control enforces an even credit distribution scheme across *active lanes*, whereas the arbiter inside the pipeline schedules each fixed-size PCIe flit in a round-robin fashion. Note that a flit is the basic transmission unit over the PCIe, which is 64B in our case. Therefore, each communication port realizes a *credit-by-credit* (or *flit-by-flit*) round-robin scheduling across all active lanes, resulting in an approximate max-min bandwidth allocation. Even though this is in contrast to the classic *bit-by-bit* round-robin (BR) [25] or *deficit round-robin* (DRR) [53] algorithm, given most PCIe



(a) Host adapter downstream. (b) PCIe Switch downstream. (c) Endpoint adapter downstream.

(d) Host adapter upstream. (e) PCIe Switch upstream. (f) Endpoint adapter upstream.

Figure 13: We report the bandwidth of two PCIe flows contending the upstream/downstream point of the host adapter, PCIe switch, and endpoint adapter, where Flow1 is fixed-size and Flow2 varies from 64B to 4KB. The number of outstanding PCIe transitions is 1.



(a) Host adapter. (b) PCIe Switch. (c) Endpoint adapter.

Figure 14: We report the P50, P90, and P99 latency of Flow1 when varying the bandwidth of Flow2, where two PCIe flows contend the host adapter, PCIe switch, and endpoint adapter, respectively. We consider the upstream and downstream of each scenario.

flows in our context contain a sequence of flits, max-min bandwidth partition is achieved.

Takeaways. In a routable PCIe fabric, any communication port (within a switch or adapter) realizes a *credit-by-credit* round-robin scheduling across different active lanes, resulting in a max-min bandwidth partition. This not only helps us to simplify the performance reasoning under traffic congestion but also assists us in deriving a predictable flow scheduler.

5.2 Fast End-to-End BW Synchronization

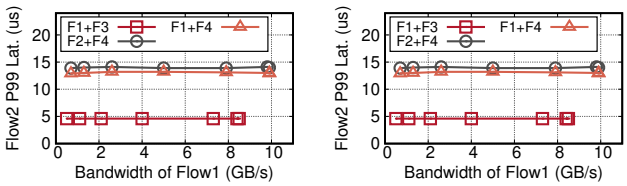
In a shared networking fabric, the link available bandwidth fluctuates considerably with the application behaviors and the underlying topological changes. Such vagaries would cause either traffic congestion (e.g., in-network queue build-up and transmission delay increase) or bandwidth underutilization. In an Ethernet fabric, the congestion control mechanism at the end host will adjust the traffic sending rate accordingly based on stipulated congestion signals. Since the routable PCIe has no such layer, in this section, we'd like to explore how PCIe flow bandwidth is adjusted based on the traffic condition.

We configure three experimental scenarios, where each has two PCIe flows sharing an intermediate communication point from different paths. The first flow is fixed and consumes more than half of the link bandwidth capacity. We then gradually increase the bandwidth utilization of the second flow and measure the P50, P90, and P99 latency of each PCIe transaction of Flow1. We find that the routable PCIe fabric has little queueing effect and the bandwidth demand can quickly propagate from the bottleneck point to upstream entities along the path. Ash shown in Figure 14-a/b, when contending the host adapter or PCIe switch, we observe up to 3.5%/2.3% or 2.6%/2.1% P99 latency increase at the upstream/downstream port. This is mainly because the flow at a congested upstream/downstream port would receive fewer credits than it requires, where such information will be back-propagated to the upstreamed ports until the source host. Since the adapter and switch within the fabric preserve little buffering, the end host could then adjust the flow rate based on how fast the PCIe transactions are delivered to the destination. However, the end host adapter (Figure 14-c) behaves differently, where contention at the upstream/downstream path can use drastic P99 latency increase, more than 10us. This is because our device engine (Figure 1) doesn't implement an auto-pacing module as the host and uses a large command queue inside, yielding significant queueing.

Takeaways. The routable PCIe fabric provides ultra-low latency communication between two endpoints and preserves little buffering at both adapters and switches. The bandwidth availability will be piggybacked over credits, which can be quickly back-propagated from the congestion point to upstream entities until the source node. One can use this as a congestion signal when coordinating concurrent flows.

5.3 Interference-free Orthogonal Paths

Last, we explore how orthogonal communication paths interact with each other over the PCIe fabric since they stay under the same PCIe root complex. As shown in Figure 12-b, we



(a) Downstream. **(b) Upstream.**

Figure 15: Performance interference among orthogonal paths.

enable concurrent data movements across these orthogonal paths (e.g., F1 v.s. F3, F2 v.s. F4, F1 v.s. F4) and explore how latency and bandwidth are affected. Specifically, we increase the bandwidth of PCIe flow1 by increasing the number of outstanding requests and measure the transaction latency of flow2 (which is a 64B flow). As shown in Figures 15, unsurprisingly, orthogonal paths across both upstream and downstream links are completely independent and interference-free.

Takeaways. Orthogonal data paths over the routable PCIe fabric can be viewed as physically isolated communication domains, imposing little performance interference. When reasoning about the fabric performance or designing flow schedulers, one can apply a divide-and-conquer strategy and categorize flows into different isolated domains.

6 Performance Model of the Routable PCIe Fabric: An Optimization Guide

Based on gathered characterization insights, we formalize the data movement problem over a routable PCIe fabric, develop an algorithm to predict the PCIe flow transmission performance, and validate its accuracy in real settings.

6.1 Problem Formalization

We describe a routable PCIe fabric as a directed tree $G = \{N, E\}$, where the host root complex is the root, PCIe endpoints are leaves, and internal/external PCIe switches are branches. Each edge represents a PCIe upstream or downstream link with capacity. The fabric holds a set of active flows $F = \{f_i\}$, where each is described by $f_i = (B_i^{in}, src, dst)$. B_i^{in} is the bandwidth of a flow when running exclusively over the fabric (i.e., standalone BW). *src* and *dst* are the source and destination nodes of a PCIe transfer, which can be a host or PCIe endpoint. We assume there is a unique path between two nodes, which is widely applicable to the PCIe subsystem.

We aim to estimate how much bandwidth a flow is allocated when deploying all the flows concurrently over the routable PCIe fabric. To achieve this, we encode the above characterization insights as the following traffic orchestration rules:

- **Rule 1: Maximum bandwidth bound.** The aggregated bandwidth of co-located flows over an upstream/downstream link should be no larger than the link bandwidth capacity (§4.3);
- **Rule 2: Bandwidth partition of a single link.** Over one PCIe link, the bandwidth partition among concurrent PCIe flows depends on the ratio of their outstanding bytes (§4.5). Besides, there exists no interference between the upstream and downstream direction (§4.6);
- **Rule 3: Approximate max-min fair bandwidth allocation.** Each communication entity guarantees the max-min fairness across active lanes/links due to the *credit-by-credit* round-robin scheduling discipline (§5.1). A PCIe flow can max out its bandwidth when the link is under-utilized and drops to a fair share when oversubscription happens;
- **Rule 4: Isolated communication domains.** There exists no interference among orthogonal paths (§5.3). One can apply it to categorize flows in the first place and then conduct performance analysis hierarchically.

Algorithm 1 Bandwidth Constraints on an Edge

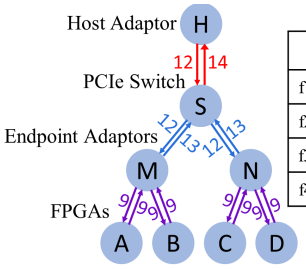
Input: Edge Capacity C, flows $F = \{f_i\}$ and their unconstrained bandwidths $\{B_i^u\}$
Output: The bandwidth constraints of flows $\{B_i^c\}$

1: **if** $\sum B_i^u \le C$ **then**
2: $B_i^c = B_i^u$, for each f_i; ▷ not oversubscribed
3: **else**
4: $n = F.size(); C' = C;$
5: **while** True **do**
6: $m = 0;$
7: **for** each f_i in F **do**
8: **if** $B_i^u \le C'/n$ **then** ▷ less than equal share
9: $B_i^c = B_i^u; C = C - B_i^u;$ ▷ not constrained
10: $F.remove(f_i); m = m + 1;$
11: $n = n - m; C' = C;$
12: **if** m == 0 **then** ▷ all flows exceed equal share
13: break; ▷ must break if oversubscribed
14: **for** each f_i in F **do**
15: $B_i^c = C'/n;$ ▷ equal share on remaining capacity

6.2 Edge Constraints Relaxing Algorithm

We propose a new algorithm (called *Edge Constraints Relaxing*) to solve the problem. The key idea is to apply iterative reduction by gradually constraining flow bandwidth based on the capacity of oversubscribed links. Given the fabric topology and deployed flows as inputs, based on the encoded rules, our algorithm first finds all the oversubscribed edges and their bandwidth constraints, and then updates each flow with its most conservative constraints. Such iterative relaxing allows all flows to converge to one allocation in finite steps where no oversubscribed edge exists. The algorithm requires us to maintain two tables: oversubscribed edges and flow constraints.

Next, we'll describe the algorithm in detail. To begin with, we first initialize all flows with their standalone bandwidths (ALG2 L1). For the oversubscribed edges table, each link is associated with its housed flows (ALG2 L2–L6). Next, for each round, the algorithm traverses each edge and determines if it is under oversubscription or not by comparing the link bandwidth capacity and the aggregated target bandwidth of its housed flows. For all oversubscribed ones, we use the Algorithm 1 based on **Rule 3** to compute the constrained bandwidth of each flow, which is then stored in the flow constraints table (ALG2 L10–L12). After all edges are traversed in this round, flows that have constraints are updated accordingly. The largest constraint of a flow is chosen as its next bandwidth (ALG2 L17). Flows not being captured means they are able to achieve their bandwidth in the current round,



Flows	Standalone
f1: H->C	8 GB/s
f2: B->D	9 GB/s
f3: H->A	7 GB/S
f4: C->D	4 GB/S

	Flows	Current	Constraints	Next
Round 1	f1	8	6, 6	6
	f2	9	6, 5	6
	f3	7	6	6
	f4	4		4

Edge	HS	SN	NC	CN	ND	BM	MS	SM	MA
Capacity	12	12	9	9	9	9	13	12	9
Flow List	f1,f3	f1,f2	f1	f4	f2,f4	f2	f3	f3	f3
Oversubscribed	T	T	F	F	T	F	F	F	F
Constraints	{6,6}	{6,6}			{5, / }				

	Flows	Current	Constraints	Next
Round 2	f1	6		6
	f2	6	5	5
	f3	6		6
	f4	4		4

Edge	HS	SN	ND
Capacity	12	12	9
Flow List	f1,f3	f1,f2	f2,f4
Oversubscribed	F	F	T
Constraints			{5, / }

(a) Testbed Topology and Input Flows **(b) Flow Constraints Table** **(c) Oversubscribed Edges Table**

Figure 16: The workflow of the proposed algorithm (§6.3). (a) shows algorithm inputs, including the topology graph and flows. (b) and (c) depict the flow constraints table and oversubscribed edges table for each execution round. The units of all numbers are GB/s.

Algorithm 2 Edge Constraints Relaxing

Input: Edges $E = \{e_i\}$ and their capacities $\{B_{e_i}\}$, flows $F = \{f_i\}$ and their standalone bandwidths $\{B_i^{in}\}$.
Output: Bandwidth allocation of flows $\{B_i^{out}\}$

1: $B_i = B_i^{in}$, for each f_i in F; ▷ initialize
2: **for** each e_i in E **do**
3: **if** e_i has flows **then**
4: EdgeTable.add(e_i); ▷ initialize Edge Table
5: **for** each f_i in e_i **do**
6: e_i.flow_list.add(f_i);
7: **while** EdgeTable.empty() == False **do**
8: **for** each e_i in E **do**
9: **if** $\sum_{f_j \in e_i} f_j > C_{e_i}$ **then** ▷ an oversubscribed edge
10: $\{B_j^c\} = Algorithm1(C_{e_i}, \{B_j\}), f_j \in e_i$;
11: **for** each $f_j \in e_i$ **do**
12: f_j.constraints.add(B_j^c);
13: **else** ▷ not oversubscribed, delete;
14: EdgeTable.delete(e_i)
15: **for** each f_i in F **do**
16: **if** f_i.constraints.empty() == False **then**
17: $B_i = f_i$.constraints.max(); ▷ flow update
18: $B_i^{out} = B_i$ ▷ loop finishes, output bandwidth

which are the final outputs and will not be updated in later phases. The insight of choosing the most conservative bandwidth constraint is that it can guarantee the flow bandwidth will always be at least its fair share. At the end of the current round, we remove all edges that are no longer oversubscribed from the edge table and use the new bandwidth (ALG2 L14) for the next round to continue the iteration. When there are no more oversubscribed edges, the bandwidths of each flow are the final allocated results (ALG2-L18).

The algorithm is guaranteed to converge because there are only finite edges generating fixed constraints in the system. At any round, every constrained flow will be reduced, indicating that at least one constraint will be eradicated from the list. Thus, the algorithm must converge in finite rounds. Since the number of edges a flow traverses is the maximum number of constraints, the number of steps to converge is bounded by twice the height of the tree. Our algorithm has a $O(N)$ complexity where N is the number of input flows.

6.3 A Walkthrough Example

We now use an example to show how the proposed algorithm works. Figure 16-a shows a PCIe tree topology based on our testbed. Specifically, node H represents the host adapter, node S refers to the external PCIe switch, and nodes M/N are endpoint adapters. Nodes A/B are two FPGAs in chassis 1 and C/D are the other two in chassis 2. The flows and their standalone bandwidths are listed in the right table.

At the beginning, we initialize the flows with standalone bandwidth and fill the flow list in the edge table. Then, we calculate the aggregated bandwidth of each edge and compare it with the link capacity to determine oversubscribed ones. For example, edge HS is an oversubscribed one because its capacity is 12, while the two housed flows f1 and f3 require 8 and 9 transmission bandwidths ($8+9>12$), respectively. Applying the Algorithm 1 to HS, we will obtain a constraint {6,6}, which is inserted into f1 and f3's constraints in the table (Figure 16-b). In the first round, the algorithm decides that edges HS, SN, and ND are oversubscribed ones. As shown in Figure 16-b, HS and SN put 6,6 to the f1's constraints entry, SN and ND insert 6,5 to the f2's constraints, and HS writes 6 to f3's constraints. After finding all constraints, we now use these constrained bandwidths to reduce flows. We update the f1's bandwidth to 6 as its largest constraint is 6. The same logic is applied to f2 and f3. Since f4 has no constraints, as discussed above, it means that f4 can take the original bandwidth as the final one with no bandwidth reduction.

In the second round, only those oversubscribed edges are left in the edge table (Figure 16-c). After the first round, HS and SN are no longer oversubscribed links, except ND. We will then repeat the same procedure to update the flow. The entire process is completed at the end of the second round as all links are not oversubscribed (i.e., the edge table becomes empty). So our final estimated results are: f1, f2, f3, and f4 will achieve 6, 5, 6, 4, respectively.

6.4 Validation and Discussion

We designed three experiments to validate the accuracy of our proposed algorithms. Each experiment targets different oversubscribed links. We use rPCIeBench to figure out the standalone bandwidth of each flow and the link capacity (Table 2-a). Tables 2-b/c/d present the comparison of each experimental scenario (i.e., measured v.s. estimated).

The oversubscribed edge of the first validation experiment is HS, where two downstream PCIe flows from the host fully utilize the bandwidth between the host and switch. The two flows should receive an equal bandwidth share. As shown in Table 2-b, H→C and H→A achieve 5.77 GB/s and 5.43

Edge	Capacity
HS / SH	11.55 / 12.25
SM, SN	15.56
MS, NS	15.46
MA,MB,NC,ND	8.74
AM,BM,CN,DN	11.70

(a) Measured edge capacities.

Flow	Sta.	Est.	Mea.
H->C	8.51	5.78	5.77
H->A	7.21	5.78	5.43
C->B	1.76	1.76	1.70
B->D	7.19	7.19	6.93
A->H	2.54	2.54	2.52

(b) HS is oversubscribed.

Flow	Sta.	Est.	Mea.
H->C	8.56	8.37	7.82
H->A	0.53	0.53	0.47
C->B	1.76	1.76	1.63
B->D	7.19	7.19	7.00
A->H	2.54	2.54	2.53

(c) SN is oversubscribed.

Flow	Sta.	Est.	Mea.
H->C	0.55	0.55	0.49
H->D	8.58	4.37	3.58
C->B	1.76	1.76	1.75
B->D	7.19	4.37	3.59
A->H	2.54	2.54	2.54

(d) ND is oversubscribed.

Table 2: Measured bandwidth v.s. estimated bandwidth for three validation experiments. *Sta.* refers to the standalone bandwidth measured via rPCIeBench. *Est.* means the estimated bandwidth using our Algorithm 2. *Mea.* shows the actual measured bandwidth when all flows are deployed. The unit of all numbers is GB/s.

GB/s, respectively, close to our estimation. SN link is not oversubscribed after H→C is constrained. The average error of our estimation is 2.94%. In the second validation experiment, the oversubscribed edge is SN. Our algorithm suggests that H→C should be reduced for fairness, same as the measured result (Table 2-c). Yet all other flows are affected a little bit. Our modeling indicates that most PCIe flows in this setting have no interactions with each other. But still, the algorithm identifies the most constrained flow (H→C) and delivers a 5.15% estimation error. In the last validation experiment, the oversubscribed point is at edge ND. The computed allocation suggests an equal bandwidth share should happen on the endpoint link while other flows are left unchanged. The actual bandwidth (Table 2-d) is almost the same except the overall link capacity on ND decreases. We suspect this is mainly due to the MMIO contention impact, which bounds the maximum PCIe bandwidth [50]. Because of this, our algorithm is able to predict the right trends, but the estimation error rate is increased to 11.32% due to decreased link capacity.

7 Related Work

PCIe Characterization. People have studied extensively on understanding PCIe for different contexts. Kalia *et al.* [35] explored the interaction between PCIe and RDMA primitives, providing a low-level evaluation and system design guidelines to optimize RDMA-based systems. Researchers [50] proposed a theoretical model of PCIe and developed the `pcie-bench` to systematically measure the host PCIe substrate. NetTLP [37] enhances the observability of PCIe transactions by separating the PCIe transaction layer into a software layer and connecting it to the hardware root complex. Wei *et al.* [56] characterized an off-path SmartNIC when running distributed applications and unearthed the peculiarities of the SmartNIC PCIe subsystem. Unlike these studies that predominantly consider intra-host PCIe, we focus on understanding the performance implications of routable PCIe when holding composable infrastructures.

System Benchmarking. Our study benefits from prior pioneering efforts in developing benchmarking systems for different computing domains, such as single-/multi-core processors [21, 33], domain-specific accelerators [32, 41, 52], cloud applications [26, 36], microservices/serverless functions [28, 54], interconnects [35, 50], storage systems [22, 31, 43, 48, 49], and programmable networking devices [27, 30, 44, 46, 51, 56, 58, 59]. We follow similar design principles when building the rPCIeBench framework: hardware/software open-source across the system stack, end-to-end visibility, elastic modularity for upgrading/replacing sub-components, and parameterized deployments with reconfigurability.

Memory Fabrics. The past few years have seen rising interest from industry [3–8, 14, 19, 20] and academia [29, 34, 42, 45, 47, 55, 57] in developing this new cluster interconnect. Memory fabrics (such as CXL [8] and CCIX [7]), providing the load/store interface, allow tight integration of cross-server computational resources, yielding next-generation system composability. However, under the hood, the memory load/store instructions are carried over a PCIe-like substrate. Therefore, our experimental methodology, performance analyses, and findings would be generally applicable.

Discussion. Compared with an intra-server PCIe switch, the external PCIe one offers higher scalability, allows elastic resource management, and can assign remote endpoint PCIe devices to different server hosts. However, its routing table is still constructed during the bus enumeration phase when booting the server host. Our characterization results and findings (such as max-min bandwidth fairness and fast bandwidth synchronization) are applicable to other routable PCIe testbeds, not only GigaIO Fabrex. Future PCIe Gen5/6 devices would see a latency and throughput improvement.

8 Conclusion

This paper presents **rPCIeBench**, a software-hardware co-designed benchmarking framework to characterize the performance of routable PCIe, the underlying cluster interconnect for building emerging composable infrastructures. Using rPCIeBench, we first examine the performance of one routable PCIe path and then dissect the in-fabric traffic orchestration behaviors. Based on the gathered insights, we develop an edge-constrained relaxing algorithm to accurately predict the communication performance of each PCIe flow over a shared routable PCIe fabric.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Anuj Kalia, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199 and CNS-2212192 and Intel CAD SRS award.

References

[1] A new twist on PCI-Express switching for the datacenter. https://www.nextplatform.com/2019/10/02/a-new-twist-on-pci-express-switching-for-the-datacenter/, 2019.

[2] Pushing PCI-Express fabrics up to the next level. https://www.nextplatform.com/2020/03/27/pushing-pci-express-fabrics-up-to-the-next-level/, 2020.

[3] Micron Compute Express Link™ (CXL™) memory expansion for the next-generation data center. https://www.micron.com/solutions/server/cxl, 2022.

[4] Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module, 2022.

[5] SK hynix Develops DDR5 DRAM CXLTM Memory to Expand the CXL Memory Ecosystem. https://news.skhynix.com/sk-hynix-develops-ddr5-dram-cxltm-memory-to-expand-the-cxl-memory-ecosystem/, 2022.

[6] The XMM CXL E3.S from SMART Modular Technologies. https://www.smartm.com/product/xmm-cxl-e3s, 2022.

[7] CCIX Specifications. https://www.ccixconsortium.com/library/specification/, 2023.

[8] CXL Specifications. https://www.computeexpresslink.org/download-the-specification, 2023.

[9] Enfabrica's Accelerated Compute Fabric. https://blog.enfabrica.net/press-release-enfabrica-raises-125-million-series-b-to-fuel-ramp-of-ai-infrastructure-networking-a8a0b21653d2, 2023.

[10] GigaIO's FabreX System. https://gigaio.com/products/fabrex-system-overview/, 2023.

[11] GroqRack Compute Cluster. https://groq.com/wp-content/uploads/2022/10/GroqRackâĎć-Compute-Cluster-Product-Brief-v1.0.pdf, 2023.

[12] H3's Falcon System. https://www.h3platform.com/solution/composable-ai, 2023.

[13] Liqid's SmartStack System. https://www.liqid.com/products/gpu-on-demand, 2023.

[14] Omega Fabric from IntelliProp. https://www.intelliprop.com/products-page, 2023.

[15] OProfile: a Statistical Profiler for Linux Systems. https://man7.org/linux/man-pages/man1/opro file.1.html, 2023.

[16] PCIe Specifications. https://pcisig.com/specifications/pciexpress/, 2023.

[17] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2023.

[18] Small Form Factor (SFF) Specifications. https://www.snia.org/technology-communities/sff/specifications, 2023.

[19] The Intel® Agilex™ 7 FPGA I-Series Development Kit. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/i-series/dev-agi027.html, 2023.

[20] The Leo Memory Accelerator Platform from Astera Labs. https://www.asteralabs.com/products/cxl-memory-platform/, 2023.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154, 2010.

[23] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, page 185–196, 1965.

[24] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, 1968.

[25] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures & Protocols*, page 1–12, 1989.

[26] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, page 37–48, 2012.

[27] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46, 2020.

[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, page 3–18, 2019.

[29] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[30] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 916–929, 2023.

[31] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, pages 1012–1027, 2023.

[32] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238, 2015.

[33] J.L. Henning. SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[34] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC'23)*, pages 585–600, 2023.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[36] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[37] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. NetTLP: A Development Platform for PCIe devices in Software Interacting with Hardware . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 141–155, 2020.

[38] HT Kung, Trevor Blackwell, and Alan Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *Proceedings of the conference on Communications architectures, protocols and applications (SIGCOMM'94)*, pages 101–114, 1994.

[39] HT Kung and Koling Chang. Receiver-oriented adaptive buffer allocation in credit-based flow control for ATM networks. In *Proceedings of INFOCOM'95*, volume 1, pages 239–252, 1995.

[40] HT Kung and Robert Morris. Credit-based flow control for ATM networks. *IEEE Network*, 9(2):40–48, 1995.

[41] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202, 2018.

[42] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[43] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 83–90, 2018.

[44] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[45] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[46] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, pages 318–333. 2019.

[47] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[48] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, pages 106–122, 2021.

[49] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. {eZNS}: An elastic zoned namespace for commodity {ZNS}{SSDs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[50] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[51] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance clarity for SmartNIC offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, pages 16–22, 2020.

[52] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119, 2014.

[53] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 231–242, 1995.

[54] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[55] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023.

[56] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, 2023.

[57] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC'23)*, pages 601–617, 2023.

[58] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, 2014.

[59] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. NetFPGA: Rapid prototyping of networking devices in open source. *ACM SIGCOMM Computer Communication Review*, 45(4):363–364, 2015.

Words/Burst	Channel (#)	Throughput (GB/s)	Latency (ns)
1 (32B)	1	6.8	326.3
2 (64B)	1	13.4	330.3
4 (128)	1	13.7	649.5
8 (256B)	1	13.7	1301.9
16 (512B)	1	13.0	1677.9
1 (32B)	32	216.2	326.4
2 (64B)	32	427.4	330.5
4 (128)	32	438.2	648.7
1 (256B)	32	439.5	1297.6
1 (512B)	32	416.1	1671.5

Table 3: Throughput and latency of HBM data read when varying the number of channels.

Granularity	BRAM Latency (ns)	HBM Latency (ns)
8B	627	762
16B	632	763
32B	640	768
64B	644	766
128B	1264	1551
256B	2511	3091
512B	4993	6055
1KB	9992	11961

Table 4: MMIO read latency comparing between BRAM and HBM when varying the request size.

A HBM Performance Characterization

We characterized the latency and throughput of the enclosed HBM of U55C. Table 3 presents our results.

B BlockRAM MMIO Performance

We compared the MMIO latency between BlockRAM (BRAM) and HBM. In this experiment, we configure the BlockRAM as the target device. Table 4 presents our results.