Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD)

Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)
http://camelab.org

ABSTRACT

Compute express link (CXL) is the first open multi-protocol method to support cache coherent interconnect for different processors, accelerators, and memory device types. Even though CXL manages data coherency mainly between CPU memory spaces and memory on attached devices, we argue that it can also be useful to reform existing block storage as cost-efficient, large-scale working memory. Specifically, this paper examines three different sub-protocols of CXL from a memory expander viewpoint. It then suggests which device type can be the best option for PCIe storage to bridge its block semantics to memory-compatible, byte semantics. We then discuss how to integrate a storage-integrated memory expander into an existing system and speculate how much effect it does have on the system performance. Lastly, we visit various CXL network topologies and explore a new opportunity to efficiently manage the storage-integrated, CXL-based memory expansion.

1 INTRODUCTION

Cache coherence interconnects are recently emerged to integrate different CPUs, accelerators, and memory components into a heterogeneous, single computing domain. Specifically, the interconnect technologies maintain data coherency between CPU memory and private memory attached to devices, defining a new type of globally shared memory and network space. While there have been several efforts to coherently connect different hardware components, such as Gen-Z [1] and CCIX [2], *Compute Express Link* (CXL) is the first open interconnect protocol supporting various types of processors and device endpoints [3]. CXL has absorbed Gen-Z [4] and has become one of the most promising interconnect interfaces thanks to its high-speed coherence control and full compatibility with the existing bus standard, PCIe. A broad spectrum of datacenter-scale hardware such as CPU, GPU, FPGA, and domain-specific ASIC is thus expected to take significant advantage of CXL [5–7]. CXL consortium announces that it can also disaggregate memory by pooling DRAM and byte-addressable persistent memory (PMEM).

While CXL can handle diverse computing resources and memory components, it sets block storage aside and leaves a question on whether the storage can reap the benefits of CXL or not. A primary question that storage designers and system architects may have is i) *why and what can the block storage benefit from CXL?*. If there is an advantage, we should be able to answer the following questions: ii) *how can we connect the underlying block storage to the host's system memory bus?*, iii) *what kind of CXL device type should be used for the block storage and memory expander?*, and iv) *what does CXL need to improve for better utilization of the block storage?*.

In this paper, we argue that CXL is helpful in leveraging PCIe-based block storage to incarnate a large, scalable working memory by answering all the four questions mentioned above. We believe CXL is a cost-effective and practical interconnect technology that can bridge PCIe storage's block semantics to memory-compatible, byte semantics. To this end, we should carefully integrate the block storage into its interconnect network by being aware of the diversity of device types and protocols that CXL supports. This paper first discusses what a mechanism makes the PCIe storage impractical and unable to be used for a memory expander (§2). Then, we explore all the CXL device types and their protocol interfaces to answer which configuration would be the best for the PCIe storage to expand the host's CPU memory (§3).

Even though CXL can be the most promising interface for the block storage in getting closer to CPU, it is non-trivial to speculate how much effect a storage-integrated memory expander does have on system performance. As there is no CPU and fabric for CXL yet, it is also unclear for the storage designers and system architects to see how CXL-enabled storage can be implemented and interact with CPU. To answer this, we discuss what a PCIe storage device needs to change, how it can be connected to the host over CXL, and how users can access the device through load/store instructions (§4). We then project the performance of the storage-integrated memory expander by prototyping CXL agents and controllers in different FPGA nodes, all connected by a PCIe network.

HotStorage '22, June 27–28, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539745

After examining the potential benefits of converting the block semantic to byte semantic over CXL, we further discuss how to disaggregate PCIe storage resources from computing resources using CXL (§6). In particular, we visit three different network topologies and argue the pros and cons of each disaggregation option. We lastly explore a new opportunity to manage the PCIe storage efficiently at the host-side (§7). Specifically, we suggest two additional states, *determinism* and *bufferability*, and debate why these additional states can be beneficial to handle the PCIe storage in expanding the host's CPU memory with CXL.

2 WHY CXL FOR PCIE STORAGE

Byte-addressability. It is a long-standing dream for PCIe storage to have byte-addressability and be a part of working memory devices [8–17]. For example, an industry prototype and NVMe standard [18–20] offer the byte-addressability by exposing SSD's internal memory/buffer to PCIe *base address registers* (BARs). Since BARs can be directly mapped to the system memory space, the host-side kernel and applications can access the exposed memory/buffer resources just like the local memory (using load/store instructions) rather than block storage. To hide long latency imposed by SSD's backend block media (e.g., Z-NAND, Flash, Optane SSD), they can use the internal memory/buffer as a write-back inclusive cache of the backend [21–25].

Limits with non-cacheable accesses. PCIe bandwidth is fast enough to be far memory (e.g., 63GB/s∼121GB/s for Gen5/6 16× [26]). However, PCIe considers the block storage devices as just one of the peripherals that the host-side CPU needs to manage and communicate with. Thus, while the storage devices can handle load/store requests through the PCIe's BARs, they are all limited to being used as working memory in a real system. Specifically, as the memory-mapped BARs are only the interface for the host to let the underlying storage know what it requests or controls, CPU must make the load/store requests uncached and directly accessible. This non-cacheable characteristic severely degrades the performance of all memory accesses targeting the BARs. If CPU can cache/buffer the memory requests targeting the PCIe address space, there is no way for the PCIe storage to catch their arrivals. This can introduce an unexpected situation such as a system failure or storage disconnection. To prevent such a failure, x86 instruction set architectures of both Intel and AMD do not allow PCIe-related memory requests to be cached at the CPU side. Unfortunately, this nature *enforces the storage-integrated memory expanders be excluded from the conventional memory hierarchy* and *disables them from taking advantage of CPU caches*.

Compute express link. CXL is a cache coherent interconnection technology, designed initially toward supporting various accelerators and memory devices therein [3]. Specifically, CXL can offer one or more memory address spaces in the PCIe network domain coherently, which can consistently be accessed by different processors and hardware accelerators over its multi-protocol technology. The multi-protocol overrides the I/O semantic of the existing PCIe interface, thereby making all device types of CXL compatible with most existing PCIe devices, including SSDs. We will explain details of each device type in §3.

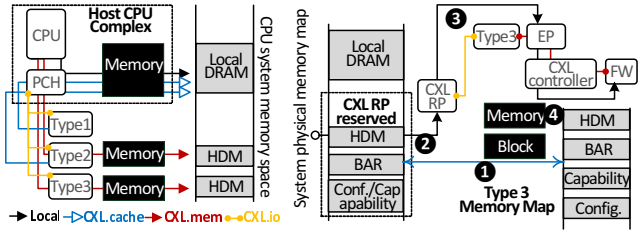
Even though CXL is built upon PCIe, it basically guarantees that all the caches across different computing complexes in the same CXL hierarchy are coherent. This can make the load/store requests (heading to the PCIe address space) *cacheable* in contrast to PCIe. The current CXL considers only DRAM or PMEM for its memory pooling, but we advocate that CXL can open a new door that changes PCIe storage's block interface to a memory-like, byte interface. As CXL's multi-protocol can integrate PCIe storage into its cache coherent memory space, it can create a much bigger memory pool than DRAM-based or PMEM-based memory expansion technologies.

3 MULTI-PROTOCOL AND CXL DEVICES

CXL provides three different sub-protocols, i) *CXL.io*, ii) *CXL.mem*, and iii) *CXL.cache*, which can also define three different types of CXL devices (*Type 1*∼*Type 3*).

Multi-protocol. CXL.io is the fundamental protocol that all CXL-attached devices and host CPUs require to communicate. Fundamentally, it employs full features of PCIe as a non-coherent load/store interface for I/O devices (e.g., device discovery/enumeration and host address configuration). To this end, CXL.io amends PCIe's hierarchical communication layers and creates a high-speed I/O channel, called *FlexBus*. FlexBus converts received CXL data to an appropriate format to leverage the physical PCIe layers (e.g., transaction, data, and link). On the other hand, CXL.cache and CXL.mem respectively add coherent cache and memory access capabilities into FlexBus, which can fan out to support multiple device domains and remote memory management. While CXL overrides PCIe, its root port (*CXL RP*) allows one or more memory addresses (exposed by the underlying CXL devices) to be mapped to a target host's cacheable system memory space. While this capability is designed toward unifying multi-domain memory devices into a single pool over coherent cache management, it can be leveraged for a memory expander using different storage technologies.

CXL device types. Based on how to combine the multi-protocol features of CXL, it declares three different device types, Type 1, Type 2, and Type 3. Figure 1a shows all the CXL device types and the protocols each device type uses.



(a) Protocols and devices. **(b) System memory mapping.**

Figure 1: CXL multi-protocol and system map.

Type 1 is CXL devices that have a local cache without employing an internal DRAM component. Type 1 is valuable if the endpoint device requires a fully coherent cache, which implies that the device can access the corresponding host's memory data through its own caches in an active manner. Most domain-specific accelerators for computationally intensive applications, such as tensor processing units [27], can be classified by this Type 1 device. Note that Type 1 devices use CXL.cache and CXL.io to manage their full cache coherence capability.

Type 2 is for discrete acceleration devices that internally employ high-performance memory modules. These memory components are referred to as *host-managed device memory* (HDM) in CXL. While the host can, in default, communicate with Type 2 devices using CXL.io (over PCIe), CXL.cache and CXL.mem are respectively used for the device to access its host-side memory and for the host to access HDM. Note that HDM differs from private memory modules employed by conventional acceleration devices such as GPUs. Even though a host can access GPU's device-side memory (e.g., GDDR), it is only performed by legacy memory copies. In contrast, a CXL-enabled host can manage HDM using cache-coherent load/store instructions. It is also expected for Type 2 devices to actively access the host's CPU memory by utilizing all the features that CXL's multi-protocol supports.

Lastly, Type 3 is designed for non-acceleration devices that only employ HDM without a processing component. CXL regulates these Type 3 devices operate primarily with CXL.mem to serve load/store requests issued by a host; it does not allow the Type 3 devices to make a request (to the host) over CXL.cache. While Type 3 does not employ CXL.cache, it can be used for expanding the host-side memory. This is because CXL.mem includes basic memory read and write interfaces for HDM. We will explain how to use HDM in §3. Note that CXL allows the Type 3 devices to manage CXL.io at the device side in an attempt to accommodate various I/O specific demands in a flexible manner.

4 INTEGRATING STORAGE INTO CXL

Device type consideration. Generally speaking, PCIe storage is not a simple, passive device. In addition to its backend's block media, PCIe storage employs large internal DRAMs to buffer/cache incoming requests and corresponding data. It also has computation capability used for diverse data processing tasks such as address translation or reliability management [28–32]. Type 2 can probably be a good option for utilizing HDM and/or integrating data processing capabilities into the storage by being aware of the semantics of host CPUs. In this paper, we however advocate Type 3 for a storage-integrated memory expander in CXL.

There are three reasons why we believe that Type 3 devices are better than Type 2 devices for the storage-integrated memory expander. First, even though Type 2 allows the host to handle the storage-side HDM directly, Type 2 is designed for computationally intensive applications. Because of this, only one device (per CXL RP) can be connected to a host system, which makes Type 2 devices not scalable as Type 3 devices can do. Second, having full features of CXL.cache and CXL.mem can introduce another type of communication burden, thereby degrading the overall performance of the storage-integrated memory expander. Specifically, all load/store requests require checking the cache states of PCIe storage's computing complex, which exhibits multiple CXL transactions for every I/O service. Even though it is crucial for the PCIe storage to manage internal DRAM efficiently, it does not require coherently managing the host's CPU caches. Third, if we integrate a PCIe storage device into CXL as Type 2, the device should ask permission from the host whenever the storage side computing resources access its memory. This is because Type 2's CXL.cache manages both the host's local memory and HDM in a fully coherent manner, which makes the device-level I/O performance even worse than before.

Storage-side modification. PCIe storage devices typically employ a PCIe endpoint and NVMe controllers to parse the incoming requests and transfer data between a host and SSD's internal DRAMs [33, 34]. Thus, a hardware change at the storage side can be simple, which in turn makes most storage devices easily support Type 3 with a minor modification. For example, we can compose a CXL storage controller to handle CXL transaction packet formatting and CXL.io control by leveraging the existing PCIe endpoint logic. Similarly, the existing NVMe controller's capabilities, such as command parsing and page memory copies, can be simplified to implement the read and write interfaces of CXL.mem. Note that the NVMe specification allows PCIe storage to realize its controllers in either firmware or hardware [35, 36]. However, we believe it is better to automate the service routine of CXL.mem's reads and writes over hardware while letting firmware manage the internal DRAMs and the back-end block media.

System integration. Figure 1b shows how CXL can connect a PCIe storage device to a host and explains how the host-side users directly access the storage device through load/store

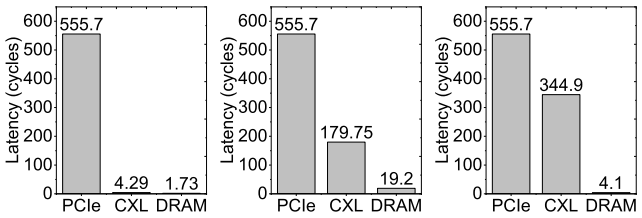
instructions. In this example, the host's system bus employs a CXL RP connecting a PCIe storage device as Type 3; we will discuss a system option to disaggregate many storage devices from the host resources in §6. When the host boots, it enumerates CXL devices connected to its RP and initializes the devices by mapping their internal memory spaces to the system memory. Specifically, the host retrieves the size of CXL BAR and HDM from the PCIe storage devices and then maps them into its system memory space (CXL RP reserved). In particular, HDM is mapped to a cacheable memory address space such that the users can access it using load/store instructions. As CXL BAR and HDM are mapped to different addresses from what a Type 3 device initially manages, the CXL RP requires letting the underlying CXL controller know where they have been mapped [❶]. The host can do this address space synchronization by writing the corresponding information (e.g., remapped address offset) to the target storage's CXL capability/configuration areas.

When an application loads or stores data on the system memory (mapped to HDM), CXL RP generates a message, called *CXL flit*, and sends it to the target's CXL storage controller via CXL.mem [❷]. The underlying endpoint and CXL controllers then parse the flit and extract the request information (e.g., command and target address) [❸]. Using the request information, the controllers can serve the data by collaborating with underlying storage firmware [❹].

5 PERFORMANCE PROJECTION

Prototype. Since there is unfortunately no processor complex that yet supports CXL.mem and CXL.io, we prototype a CXL-enabled CPU and CXL storage, each taking the role of a host and storage-integrated memory expander. Specifically, the CPU and storage are fabricated into two separate, custom FPGA boards, which are connected through a tailored PCIe backplane. We integrate CXL.mem and CXL.io agents into an in-house RISC-V CPU (64-bit O3 dual-core architecture that uses 128KB L1 and 4MB L2 caches), and 32GB OpenExpress-based NVMe storage [24] in 16nm FPGA for the host node and storage node, respectively. OpenExpress's backend media emulates Z-NAND [37] while buffering the incoming CXL requests into its internal DRAMs. In addition to this prototype (`CXL`), we evaluate a local DRAM-only system (`DRAM`) and PCIe-based memory expander (`PCIe`). `PCIe` and `CXL` use the same backend storage, but their RP's address is mapped to different places of the host's system memory.

Workloads. We use *Apex-Map*, a global memory access benchmark for large-scale computing [38]. The benchmark allows us to test underlying memory with different locality and request size levels (i.e., the parameter, α). We configure the request size as 64B, which is the same as the last-level cacheline size of our CPU. In this performance projection, we exclude time-consuming activities of internal tasks such as garbage collection; we will discuss how CXL can alleviate the long latency imposed by such internal tasks in §7. Apex-Map generates 512 million memory instructions synthetically by ranging α from 0.001 (highest locality) to 1 (lowest locality).



(a) α = 0.001. **(b) Average α.** **(c) α = 1.**

Figure 2: Performance of different memory systems.

Result analysis. Figures 2a, 2b, and 2c show each system's latency (in terms of CPU cycles) for the best case ($\alpha = 0.001$), the average case ($0.001 \leq \alpha \leq 1$), and the worst case ($\alpha = 1$), respectively. The best-case performance shows the reason why `CXL` can be more beneficial than a PCIe-based memory expander. While most memory requests in this test are hit from the CPU caches, `PCIe` cannot take any advantage of the host CPU caches, thereby exhibiting 129.5× longer latency than `CXL`. In contrast, `CXL` enjoys the CPU caches and shows excellent latency behaviors comparable with `DRAM`.

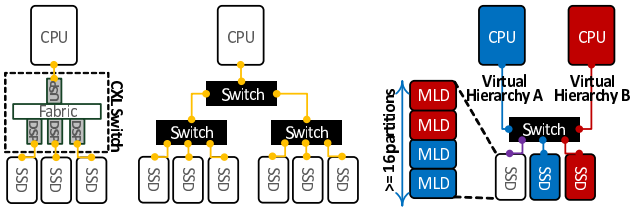
`CXL` is also better than `PCIe` by 3× for the average case. Note that, even though the performance of `CXL` is 9.3× worse than that of `DRAM`, we believe it is still in a reasonable range by considering the fact that `CXL` leverages the block storage. When there is no locality (the worst-case), `CXL` cannot hide the underlying Z-NAND latency because of the benchmark's access pattern (fully random), which exhibits 84.1× worse than `DRAM`. However, `CXL` shows still better performance compared to `PCIe` by 1.6× as it does not handle all the memory requests (on PCIe's BAR) in a synchronized fashion.

We are somewhat disappointed with the results as `CXL`'s worst-case latency characteristics are far away from DRAM's behaviors. However, most workloads exhibit high locality except for a specific application like graph processing. Considering the large capacity that the storage-integrated memory expander offers, we believe many applications can reap the benefits of `CXL`. We also believe that there is an opportunity to optimize this long latency by wisely using PCIe storage's internal DRAMs and backend block media (§7).

6 STORAGE DISAGGREGATION

This section discusses how a system can disaggregate CXL controllers and storage devices from its computing resources while keeping their byte-addressability.

Pooling storage over the byte interface. To make the interconnect network scalable, CXL 2.0 allows FlexBus to employ one or more CXL switches, each being able to have multiple



(a) Switch. **(b) Multi-switch.** **(c) Virtual hierarchy.**

Figure 3: Storage disaggregation configurations.

upstream ports (USPs) and *downstream ports* (DSPs). Even though CXL yet leaves a question undecided on how to implement a switch and its internal components, USPs and DSPs can be simply interconnected by a reconfigurable crossbar switch. Specifically, a USP can be connected to a CXL RP or another switch's DSP over FlexBus, and it internally returns the incoming message to one or more underlying DSPs as soon as possible. In contrast, a DSP links a lower-level hardware module such as a storage device's CXL endpoint or a different switch's USP. Using a switch buffer, it can control multiple CXL messages going through the DSP(s).

Figure 3a shows how a host can expand its local memory by having multiple storage devices. Specifically, each DSP connects to a different storage device, whereas a USP is linked to all the DSPs and exposes them to the host's CXL RP. For this type of storage-integrated memory expansion, the host should map each HDM to different places of its physical memory and be aware of the mapping information when the system enumerates CXL devices to configure CXL capability/configuration (BAR/HDM). While this network topology is simple enough to connect multiple PCIe storage devices, the number of lanes that a CXL switch can support is limited. Typically, a switch supports 64∼128 lanes, and thus, only 4∼8 ports are available for high-performance storage devices (using 16 lanes). In this case, as shown in Figure 3b, it can expand the host memory by adding one or more switches to the CXL network. The top switch is used to bridge the host's CXL RP and all other lower-level switches, while the leaf switches are employed to manage many PCIe storage devices. Note that the number of storage devices that a network can handle varies based on the size of the devices and the memory capacity that CXL deals with (currently, it is 4PB).

Multi-host connection management. To better utilize the storage resources, we can also connect arbitrary numbers of host CPUs to the CXL network. Since the switch's crossbar (called *fabric manager*) remembers each connection between USPs and DSPs, we can fabricate a unique routing path beginning from a host to one or more storage devices, called *virtual hierarchy* (VH). Each VH guarantees that a storage device can be mapped to a host, which is attached anywhere in the CXL network. Thus, VHs allow the system to completely disaggregate many PCIe storage devices from its multi-host computing resources for the memory expansion. While these reconfigurable VHs can realize a fully scale-out architecture, memory resources expanded by the storage devices are unfortunately tricky to control finely. Since the storage device should only be associated with a host, it can be underutilized and/or unbalanced across different CPUs.

Storage device virtualization. To address this issue, we can virtualize each storage device to be shared by different hosts. Specifically, as shown in Figure 3c, CXL allows a system to logically splits each endpoint into multiple Type 3 devices (up to 16), called *multiple logical device* (MLD). Thus, we can make each MLD define its own HDM, which can be mapped to a different place of any host of system memory, similar to a physical storage device. As each MLD associated with the same storage device can be a part of different VHs, it is expected to utilize the underlying storage resources better by allocating the memory expanders in a fine granular manner.

A disadvantage of multi-host VHs can be bandwidth sharing and/or traffic congestion. To support MLDs, PCIe storage may require partitioning the underlying backend and internal DRAMs, lowering the level of parallelism, and this can unfortunately reduce the bandwidth of each MLD. In addition, as a single storage device (and a CXL switch) can be shared by multiple hosts, the endpoint's fabric can be congested more than before. Since the performance of this multi-host memory expansion varies based on diverse perspectives and hardware configurations of CXL, it does need careful network and storage designs.

7 EXTENSION FOR STORAGE CONTROL

As CXL's Type 3 is designed for memory pooling, not block storage, there are two issues that we can further consider and discuss: i) *latency fluctuation* and ii) *data persistence*.

CXL.mem and CXL.io do not strictly manage the turnaround time of loads/stores as their CXL memory requests can be served asynchronously. However, long latency is still undesirable and can degrade the host's overall performance. For example, the PCIe storage device that we used for the performance projection assumes that there are no internal tasks (§5). The latency of internal tasks varies based on how firmware operates, but all they can make the responsiveness significantly worse than usual. In addition, if host-side libraries such as PMDK [39] insist on data persistence, the current flushing mechanism of CXL can be insufficient to handle the underlying PCIe storage. Specifically, CXL provides a *global persistent flush* (GPF) register, which enforces all data residing in the CXL network and SSD's internal DRAMs to be immediately written back to the backend media. This can also make latency behaviors of the storage-integrated memory expander(s) severely longer.

To overcome this, we suggest two simple features, i) *determinism* and ii) *bufferability*, which can be annotated to CXL messages and hint the host semantic to the underlying CXL controllers. Note that CXL allows Type 3 to manage CXL.io for diverse I/O specific demands (§3), and CXL.mem has a reserved field, which can be used for the annotation.

Latency and persistence controls. Determinism can be defined by two states, *deterministic* (DT) and *non-deterministic* (ND). DT means a host wants Type 3 devices to serve the tagged request without internal task involvement, whereas ND makes the corresponding requests fire-and-forget. For example, if DT is specified, the target storage can schedule one or more internal tasks to operate with the subsequent requests (annotated by ND) or in idles. Bufferability can also be composed by two states, *bufferable* (BF) and *non-bufferable* (NB). When BF is annotated, the corresponding requests can be cached or buffered in SSD's internal DRAMs, while the requests annotated by NB consider persistence as first-class citizens for their service. PCIe storage can then selectively write the requests back to its block media, which can avoid the situation where globally flushing all data (sitting on its large, internal DRAMs) to the block media at a time.

User scenarios. To cover diverse user scenarios, determinism, bufferability, and GPF can be used in either an individual or a combination (e.g., BF+DT, BF+ND, NB+DT, and NB+ND). For example, databases and transactional memories (e.g., `libpmem` and `libpmemobj`) log the data when a transaction begins. Since the data associated with the log is not necessary to be persistent before its commit, the host can log the transactions with either BF+ND or NB+ND. During this time, the storage can secure a time to perform internal tasks by buffering all incoming writes. When there is a transaction commit, it can configure GPF to flush all buffered data and write the commit (if needed) with NB+DT.

Since an operator of most instructions waits for its operand arrivals, loads can typically take advantage of DT. However, if there is no subsequent instruction that uses a result of the instruction issued in the previous (i.e., read-after-write dependency), the current loads do not need to be synchronized. We can thus precisely use BF+DT or BF+ND for the loads, which allows the storage to prefetch the data into its internal DRAMs. For example, since data are somewhat engaged with spatial/temporal localities in a loop code segment (e.g., matrix calculation), we can let the storage know that the data will be hit by the internal DRAMs sooner or later again.

Another use scenario to take advantage of the annotation is lock and synchronization management. Their mechanisms (e.g., spin, fence, and barrier) do not need persistence in most cases, but the latency is the matter. For example, a spinlock uses an atomic instruction such as compare-and-swap or compare-and-exchange, which is composed of a memory read and a write. While the spinlock does not place its parameters in the CPU cache, the corresponding atomic instruction keeps iterating to access the same memory address. Thus, it would be better for both the loads/stores to access Type 3 devices with BF+DT. Memory fences and barriers are also similar to the spinlock as their lifetime is bounded to the running process rather than the data.

8 CONCLUSION AND FUTURE WORK

This paper examines CXL from a memory expander viewpoint and explores different configurations to transfer PCIe's block semantic to memory-compatible byte semantic. As our performance projection is imperfect and limited to studying diverse perspectives of a storage-integrated expander, we consider extending this work by accommodating various software and hardware environments. We also believe that the several characteristics of CXL-based memory expansion that this paper discussed will lead to many architectural changes in both software and hardware, which can be worthwhile to study in the near future.

REFERENCES

[1] Gen-Z Consortium. Gen-Z Final Specifications. https://genzconsortium.org/specifications/.

[2] CCIX Consortium. CCIX Base Specification 1.1. https://www.ccixconsortium.com/library/specification/.

[3] CXL Consortium. Compute Express Link Specification Revision 2.0. https://www.computeexpresslink.org/download-the-specification.

[4] Gen-Z Consortium. Exploring the Future: CXL Consortium and Gen-Z Consortium Sign Letter of Intent to Advance Interconnect Technology. https://bit.ly/3tXPIod.

[5] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[6] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[7] Zixuan Wang, Joonseop Sim, Euicheol Lim, and Jishen Zhao. Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems. In *Proceedings of The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[8] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Nam Sung Kim, Mahmut Taylan Kandemir, and Myoungsoo Jung. Revamping storage class memory with hardware automated memory-over-storage solution. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[9] Changmin Lee, Wonjae Shin, Dae Jeong Kim, Yongjun Yu, Sung-Joon Kim, Taekyeong Ko, Deokho Seo, Jongmin Park, Kwanghee Lee, Seongho Choi, Namhyung Kim, Vishak G, Arun George, Vishwas V, Donghun Lee, Kangwoo Choi, Changbin Song, Dohan Kim, Insu Choi, Ilgyu Jung, Yong Ho Song, and Jinman Han. Nvdimm-c: A byte-addressable non-volatile memory module for compatibility with

standard ddr memory interfaces. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[10] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[11] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write dependency disentanglement with {HORAE}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[12] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[13] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped {I/O} for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC)*.

[14] Anirudh Badam and Vivek S Pai. {SSDAlloc}: Hybrid {SSD/RAM} memory management made easy. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[15] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST)*.

[16] S Kazama, S Gokita, S Kuwamura, E Yoshida, J Ogawa, and Y Honda. Memory expansion technology using software-controlled ssd. In *Proc. Flash Memory Summit*.

[17] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. *Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores*. 2020.

[18] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[19] Chander Chadha. NVMe SSD with Persistent Memory Region . shorturl. at/hrPS3.

[20] Stephan Bates. Enabling Remote Access to Persitent Memory on an I/O Subsystem using NVMe and RDMA. https://tinyurl.com/2jykndr6.

[21] Jaeho Kim, Donghee Lee, and Sam H Noh. Towards {SLO} complying {SSDs} through {OPS} isolation. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[22] Bryan S Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for {SSD} reliability. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[23] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[24] Myoungsoo Jung. {OpenExpress}: Fully hardware automated open research framework for future fast {NVMe} devices. In *2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[25] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[26] PCISIG. PCI Express 6.0 Specification. https://pcisig.com/pci-express-6.0-specification.

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[28] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving {SSD} erase costs using {WOM} codes. In *13th USENIX Conference on File and Storage Technologies (FAST)*.

[29] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards {Energy-Efficient},{In-Situ} data analytics on {Extreme-Scale} machines. In *11th USENIX Conference on File and Storage Technologies (FAST)*.

[30] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. {DC-Store}: Eliminating noisy neighbor containers using deterministic {I/O} performance and resource isolation. In *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[31] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. Austere flash caching with deduplication and compression. In *2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[32] Mohit Saxena, Yiying Zhang, Michael M Swift, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Getting real: Lessons in transitioning research simulations into hardware systems. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[33] Jie Zhang and Myoungsoo Jung. Flashabacus: a self-governing flash-based accelerator for low-power systems. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[34] Myoungsoo Jung. Exploring design challenges in getting solid state drives closer to cpu. *IEEE Transactions on Computers*, 2014.

[35] NVM Express Work Group. NVMe Base Specification. https://nvmexpress.org/developers/nvme-specification/.

[36] Gyuyoung Park and Myoungsoo Jung. Automatic-ssd: full hardware automation over new memory for high performance and energy efficient pcie storage cards. In *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020.

[37] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D.G. Lee, Jin-Hyeok Choi, and Jaeheon Jeong. A flash memory controller for 15us ultra-low-latency ssd using high-speed 3d nand flash with 3us read time. In *2018 IEEE International Solid State Circuits Conference (ISSCC)*, 2018.

[38] E. Strohmaier and Hongzhang Shan. Apex-map: A global data access benchmark to analyze hpc systems and parallel programming paradigms (sc). In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[39] Piotr Balcer. Persistent Memory Development Kit. https://pmem.io/pmdk/.