Memory Pooling With CXL

Donghyun Gouk, Miryeong Kwon, and Hanyeoreum Bae, *KAIST, Daejeon, 34141, South Korea*
Sangwon Lee and Myoungsoo Jung, *KAIST and Panmnesia, Daejeon, 34141, South Korea*

Compute Express Link (CXL) has recently attracted great attention thanks to its excellent hardware heterogeneity management and resource disaggregation capabilities. Even though there is yet no commercially available product or platform integrating CXL into memory pooling, it is expected to make memory resources practically and efficiently disaggregated much better than ever before. In this article, we propose directly accessible memory disaggregation, DirectCXL that straight connects a host processor complex and remote memory resources over CXL's memory protocol (CXL.mem). Our empirical evaluation shows that DirectCXL exhibits around 7× better performance than remote direct memory access (RDMA)-based memory pooling for diverse real-world workloads.

Memory pooling (i.e., disaggregation) has attracted great attention thanks to its high memory utilization, transparent elasticity, and resource management efficiency.[1,2] Many studies have explored various software and hardware approaches to realize memory pooling and put significant efforts into making it practical in large-scale systems.[3,4,5,6]

We can broadly classify the existing memory pooling runtimes into two different approaches based on how they manage data between a host and memory server(s): 1) page-based and 2) object-based. The page-based approach[3,4] utilizes virtual memory techniques to use disaggregated memory without a code change. It swaps page cache data residing on the host's local dynamic random-access memorys (DRAMs) from/to the remote memory systems over a network in cases of a page fault. On the other hand, the object-based approach handles disaggregated memory from a remote using their own database, such as a key-value store (KVS) instead of leveraging the virtual memory systems.[5,6] This approach can address the challenges imposed by address translation (e.g., page faults, context switching, and write amplification), but it requires significant source-level modifications and interface changes.

While there are many variants, all the existing approaches need to move data from the remote memory to the host memory over remote direct memory access (RDMA)[5,6] (or similar fine-grain network interfaces[3,4]). In addition, they even require managing locally cached data in either the host or memory nodes. Unfortunately, the data movement and its accompanying operations (e.g., page cache management) introduce redundant memory copies and software fabric intervention, which makes the latency of disaggregated memory longer than that of local DRAM accesses by multiple orders of magnitude. In this work, we advocate Compute Express Link (CXL), which is a new concept of open industry standard interconnects offering high-performance connectivity among multiple host processors, hardware accelerators, and input/output (I/O) devices. CXL is originally designed to achieve the excellency of heterogeneity management across different processor complexes, but both industry and academia anticipate its cache coherence ability can help improve memory utilization and alleviate memory overprovisioning with low latency. Even though CXL exhibits a great potential to realize memory pooling with low monetary cost and high performance, it has not been yet made for production, and there is no platform to integrate memory into a memory pooling network.

We demonstrate DirectCXL, direct accessible disaggregated memory that connects host processor complex and remote memory resources over CXL's memory protocol (CXL.mem). To this end, we explore a practical design for CXL-based memory pooling and make it real. Specifically, we first show how to disaggregate memory over CXL and integrate the disaggregated memory into processor-side system memory. This includes implementing CXL controller(s) that

0272-1732 © 2023 IEEE
Digital Object Identifier 10.1109/MM.2023.3237491
Date of publication 16 January 2023; date of current version 13 March 2023.

employs multiple DRAM modules on a remote side. We then prototype a set of network infrastructure components, such as a CXL switch, to make the disaggregated memory connected to the host in a scalable manner. As there is no operating system that supports CXL, we also offer CXL software runtime that allows users to utilize the underlying disaggregated memory resources through sheer load/store instructions. DirectCXL does not require any data copies between the host memory and remote memory, and therefore, it can expose the true performance of remote-side disaggregated memory resources to the users.

IN THIS WORK, WE BRING CXL 2.0 INTO A REAL SYSTEM AND ANALYZE THE PERFORMANCE CHARACTERISTICS OF A CXL-ENABLED DISAGGREGATED MEMORY DESIGN. WE ALSO DISCUSS A COMPOSABLE SERVER ARCHITECTURE THAT CAN BE IMPLEMENTED BY A FUTURE CXL PROTOCOL.

We prototype DirectCXL using many customized memory add-in-cards, 16-nm field-programmable gate array (FPGA)-based processor nodes, a switch, and a PCIe backplane. On the other hand, DirectCXL software runtime is implemented based on Linux 5.13. In this work, we bring CXL 2.0 into a real system and analyze the performance characteristics of a CXL-enabled disaggregated memory design. We also discuss a composable server architecture that can be implemented by a future CXL protocol in the "Toward Composable Server Architecture" section.

- Realizing CXL network infrastructure from the beginning to the end.
- Software runtime design and implementation for memory pooling.
- Prototyping memory expanders and analyzing pooling system with real-world applications.
- CXL-based composable system architecture analysis and discussion.

The results of our real system evaluation show that the disaggregated memory resources of DirectCXL can exhibit DRAM-like performance when the workload can enjoy the host processor's cache. When the load/store instructions go through the CXL network and are served from the disaggregated memory, DirectCXL's latency is shorter than the best latency of RDMA by 6.2×, on average. For real-world applications, DirectCXL exhibits 7× better performance than RDMA-based memory pooling, on average.

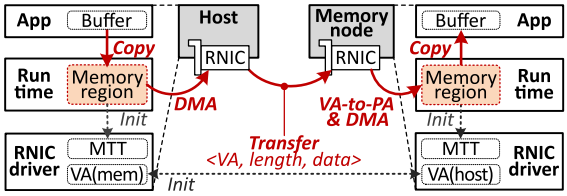

FIGURE 1. Data movement over RDMA.

MEMORY POOLING AND RELATED WORK

Remote Direct Memory Access

The basic idea of memory pooling is to connect a host with one or more memory nodes, such that it does not restrict a given job execution due to limited local memory space. For the backend network control, most disaggregation work employ RDMA[5,6] or similar customized direct memory access (DMA) protocols.[3,4] Figure 1 shows how RDMA-style data transfers (one-sided RDMA) work. For both the host and memory node sides, RDMA needs hardware support, such as RDMA NIC (RNIC), which is designed toward removing the intervention of the network software stack as much as possible. To move data between them, processes on each side first require defining one or more memory regions (MRs) and letting the MR(s) to the underlying RNIC. During this time, the RNIC driver checks all physical addresses associated with the MR's pages and registers them to RNIC's memory translation table (MTT). Since those two RNICs also exchange their MR's virtual address at the initialization, the host can simply send the memory node's destination virtual address with data for a write. The remote node then translates the address by referring to its MTT and copies the incoming data to the target location of MR. Reads over RDMA can also be performed in a similar manner. Note that, in addition to the memory copy operations (for DMA), each side's application needs to prepare or retrieve the data into/from MRs for the data transfers, introducing additional data copies within their local DRAM.[7]

Swap: Page-Based Memory Pool

Page-based memory pooling[3,4] achieves memory elasticity by relying on virtual memory systems. Specifically, this approach intercepts paging requests when there is a page fault, and then it swaps the data to a remote memory node instead of the underlying storage. To this end, a disaggregation driver underneath

the host's kernel swap daemon (kswapd) converts the incoming block address to the memory node's virtual address. It then copies the target page to RNIC's MR and issues the corresponding RDMA request to the memory node. Since all operations for memory disaggregation is managed under kswapd, it is easy-to-adopt and transparent to all user applications. However, page-based systems suffer from performance degradation due to the overhead of page fault handling, I/O amplifications, and context switching when there are excessive requests for the remote memory.[6]

KVS: Object-Based Memory Pool

In contrast, object-based memory disaggregation systems[5,6] directly intervene in RDMA data transfers using their own database, such as KVS. Object-based systems create two MRs for both host and memory node sides, each dealing with buffer data and submission/completion queues (SQ/CQ). Generally, they employ a KV hashtable whose entries point to corresponding (remote) memory objects. Whenever there is a request of Put (or Get) from an application, the systems place the corresponding value into the host's buffer MR and submit it by writing the remote side of SQ MR over RDMA. Since the memory node keeps polling SQ MR, it can recognize the request. The memory node then reads the host's buffer MR, copies the value to its buffer MR over RDMA, and completes the request by writing the host's CQ MR. As it does not lean on virtual memory systems, object-based systems can address the overhead imposed by page swap. However, the performance of object-based systems varies based on the semantics of applications compared to page-based systems; kswapd fully utilizes local page caches, but KVS does not for remote accesses. In addition, this approach is unfortunately limited because it requires significant source-level modifications for legacy applications.

DIRECT-ACCESS MEMORY DISAGGREGATION

While caching pages and network-based data exchange are essential in the current technologies, they can unfortunately significantly deteriorate the performance of memory disaggregation. DirectCXL instead directly connects remote memory resources to the host's computing complex and allows users to access them through sheer load/store instructions.

Connecting Host and Memory Over CXL

CXL devices and controllers: In practice, existing memory disaggregation techniques still require computing resources at the remote memory node side. This is because all DRAM modules and their interfaces are designed as passive peripherals, which require the control computing resources. CXL.mem in contrast allows the host computing resources directly access the underlying memory through PCIe buses (FlexBus); it works similar to local DRAM, connected to their system buses. Thus, we design and implement CXL devices as pure passive modules, each being able to have many DRAM dual inline memory modules (DIMMs) with its own hardware controllers. Our CXL device employs multiple DRAM controllers, connecting DRAM DIMMs over the conventional double data rate (DDR) interfaces. Its CXL controller then exposes the internal DRAM modules to FlexBus through many PCIe lanes. In the current architecture, the device's CXL controller parses incoming PCIe-based CXL packets, called CXL flits, converts their information (address and length) to DRAM requests, and serves them from the underlying DRAMs using the DRAM controllers.

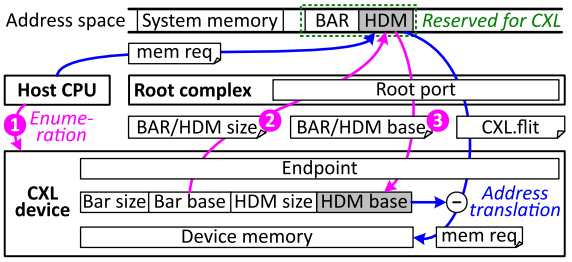

FIGURE 2. DirectCXL's connection method.

Integrating devices into system memory: Figure 2 shows how CXL devices' internal DRAMs are mapped (exposed) to a host's memory space over CXL. The host CPU's system bus contains one or more CXL root ports (RPs), which connect one or more CXL devices as endpoint (EP) devices. Our host-side kernel driver first enumerates CXL devices by querying the size of their base address register (BAR) and their internal memory, called host-managed device memory (HDM), through PCIe transactions. Based on the retrieved sizes, the kernel driver maps BAR and HDM in the host's reserved system memory space and lets the underlying CXL devices know where their BAR and HDM (base addresses) are mapped in the host's system memory. When the host CPU accesses an HDM system memory through load/store instruction, the request is delivered to the corresponding RP, and the RP converts the requests to a CXL flit. Since HDM is mapped to a different location of the system memory, the memory address space of HDM is different from that of EP's internal DRAMs. Thus, the CXL controller translates the incoming addresses by simply deducting HDM's base address from them and issues the translated request to the underlying DRAM controllers.

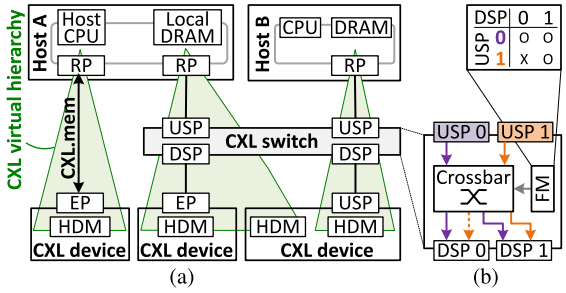

FIGURE 3. DirectCXL's network and switch. (a) CXL virtual hierarchy. (b) CXL switch.

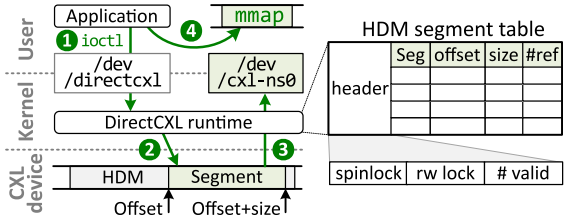

FIGURE 4. DirectCXL software runtime.

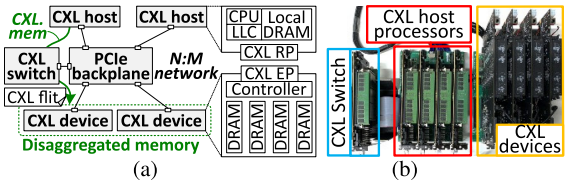

FIGURE 5. CXL-enabled cluster. (a) Network topology. (b) Implementation.

The results are returned to the host via a CXL switch and FlexBus. Note that, since HDM accesses have no software intervention or memory data copies, DirectCXL can expose the CXL device's memory resources to the host with low access latency.

Designing CXL network switch: Figure 3(a) illustrates how DirectCXL can disaggregate memory resources from a host using one or more and CXL devices, and Figure 3(b) shows our CXL switch organization therein. The host's CXL RP is connected to upstream port (USP) of either a CXL switch or the CXL device directly. The CXL switch's downstream port (DSP) also connects either another CXL switch's USP or the CXL device. Note that our CXL switch employs multiple USPs and DSPs. By setting an internal routing table, our CXL switch's fabric manager (FM) reconfigures the switch's crossbar to connect each USP to a different DSP, which creates a virtual hierarchy from a root (host) to a terminal (CXL device). Since a CXL device can employ one or more controllers and many DRAMs, it can also define multiple logical devices, each exposing its own HDM to a host. Thus, different hosts can be connected to a CXL switch and a CXL device. Note that each CXL virtual hierarchy only offers the path from one to another to ensure that no host is sharing an HDM.

Software Runtime for DirectCXL

In contrast to RDMA, once a virtual hierarchy is established between a host and CXL device(s), applications running on the host can directly access the CXL device by referring to HDM's memory space. However, it requires software runtime/driver to manage the underlying CXL devices and expose their HDM in the application's memory space. We thus support DirectCXL runtime that simply splits the address space of HDM into segments, called cxl-namespace. DirectCXL runtime then allows the applications to access each CXL-namespace as memory-mapped files (`mmap`).

Figure 4 shows the software stack of our runtime and how the application can use the disaggregated memory through cxl-namespaces. When a CXL device is detected (at a PCIe enumeration time), DirectCXL driver creates an entry device (e.g., `/dev/directcxl`) to allow users to manage a cxl-namespace via `ioctl`. If users ask a cxl-namespace to the entry device, the driver checks a (physically) contiguous address space on an HDM by referring to its HDM segment table whose entry includes a segment's offset, size, and reference count (recording how many cxl-namespaces that indicate this segment). Since multiple processes can access this table, its header also keeps necessary information, such as spinlock, read/write locks, and a summary of table entries (e.g., valid entry numbers). Once DirectCXL driver allocates a segment based on the user request, it creates a device for `mmap` (e.g., `/dev/cxl-ns0`) and updates the segment table. The user application can then map the cxl-namespace to its process virtual memory space using `mmap` with `vm_area_struct`.

Prototype Implementation

Figure 5(a) illustrates our design of a CXL network topology to disaggregate memory resources, and the corresponding implementation in a real system is shown in Figure 5(b). There are four compute hosts connected to four CXL devices through a CXL switch in our prototype, but those numbers can scale by having more CXL switches. Specifically, each CXL device prototype is built on our customized CXL memory blade that employs 16-nm FPGA and eight DDR4 DRAM modules (64 GB). In the FPGA, we fabricate a CXL controller and eight DRAM controllers, each managing the CXL endpoint and internal DRAM channels. As yet there is no processor architecture supporting CXL, we also build

our own in-house host processor using RISC-V ISAs, which employs four out-of-order cores whose last-level cache (LLC) implements CXL RP. Each CXL-enabled host processor is implemented in a high-performance datacenter accelerator card, taking a role of a host, which can individually run Linux 5.13 and DirectCXL software runtime. We expose CXL devices to the hosts through our PCIe backplane. We extended the backplane with one more accelerator card that implements DirectCXL's CXL switch. This switch implements FM that can create multiple virtual hierarchies, each connecting a host and a CXL device in a flexible manner.

To the best of our knowledge, there are no commercialized CXL 2.0 IPs for the processor side's CXL engines and CXL switch. Thus, we built all DirectCXL IPs from the ground. Our in-house softcore processors work at 100 MHz while CXL and PCIe IPs (RP, EP, and Switch) operate at 250 MHz. Very recently, CXL 3.0 has just been released, which will be analyzed in the "Toward Composable Server Architecture" section with an introduction of our on-going project.

EVALUATION

Testbed prototypes for memory disaggregation: In addition to the CXL environment that we implemented in the "Prototype Implementation" section (`DirectCXL`), we set up the same configuration with it for our RDMA-enabled hardware system (`RDMA`). For `RDMA`, we use Mellanox ConnectX-3 VPI InfiniBand RNIC instead of our CXL switch as RDMA network interface card (RNIC). In addition, we port Mellanox OpenFabric Enterprise Distribution (OFED) v4.9 as an RDMA driver to enable RNIC in our evaluation testbed. Finally, we port FastSwap[1] and HERD[5] into RISC-V Linux 5.13.19 computing environment atop `RDMA`, each realizing page-based disaggregation (`Swap`) and object-based disaggregation (`KVS`).

For better comparison, we also configure the host processors to use only their local DRAM (`Local`) by disabling all the CXL memory nodes. Note that we used the same testbed hardware mentioned previously for both CXL experiments and non-CXL experiments but differently configured the testbed for each reference. For example, our testbed's FPGA chips for the host (in-house) processors and CXL devices use all the same architecture/technology and product lineup.

Benchmark and workloads: Since there is no microbenchmark that we can compare different memory pooling technologies (`RDMA` versus `DirectCXL`), we also build an in-house memory benchmark for in-depth analysis of those two technologies (see the "In-depth Analysis of RDMA and CXL" section). For `RDMA`, this benchmark allocates a large size of the memory pool at the remote side in advance. This benchmark allows a host processor to send random memory requests to a remote node with varying lengths; the remote node serves the requests using the preallocated memory pool. For `DirectCXL` and `Local`, the benchmark maps `cxl namespace` or `anonymous mmap` to user spaces, respectively. The benchmark then generates a group of RISC-V memory instructions, which can cover a given address length in a random pattern and directly issues them without software intervention. For the real workloads, we use Facebook's deep learning recommendation model (DLRM), an in-memory database used for the HERD evaluation (MemDB[5]), and four graph analysis workloads (MIS, BFS, CC, and BC) coming from Ligra.[8] Table 1 summarizes the per-node memory usage for each workload that we tested.

TABLE 1. Memory usage characteristic.

	Per-node usage		Data stored in remote memory
	Local	Remote	
DLRM	Less than 100 MB	17 GB	Embedding tables
MemDB		4 GB	Key-value pairs and tree structure
Ligra		7 GB	Deserialized graph structure

In-Depth Analysis of RDMA and CXL

In this section, we compare the performance of RDMA and CXL technologies when the host and memory nodes are configured through a 1:1 connection. Figure 6(a) shows latency breakdown of `RDMA` and `DirectCXL` when reading 64 bytes of data. One can observe from the figure that `RDMA` requires two DMA operations, which doubles the PCIe transfer and memory access latency. In addition, the communication overhead of InfiniBand (`Network`) takes 78.7% (2,129 cycles) of the total latency (2,705 cycles). In contrast, `DirectCXL` only takes 328 cycles for memory load request, which is 8.3× faster than `RDMA`. There are two reasons behind this performance difference. First, `DirectCXL` straight connects the compute nodes and memory nodes using PCIe while `RDMA` requires protocol/interface changes between InfiniBand and PCIe. Second, `DirectCXL` can translate memory load/store request from LLC into the CXL flits whereas `RDMA` must use DMA to read/write data from/to memory.

Sensitivity tests: Figure 6(b) decomposes `RDMA` latency into essential hardware (`Memory` and `Network`), software (`Library`), and data transfer latencies (`Copy`). In this evaluation, we instrument two user-level InfiniBand libraries, `libibverbs` and `libmlx4` to measure the software side latency. `Library` is the primary performance bottleneck in

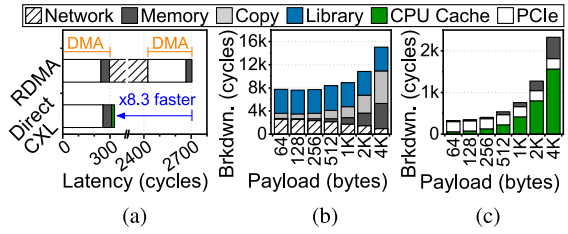

FIGURE 6. Latency breakdown of RDMA and CXL. (a) Hardware only. (b) RDMA. (c) DIRECTCXL.

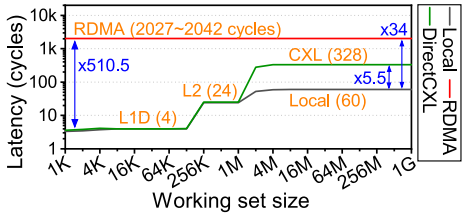

FIGURE 7. Memory hierarchy performance.

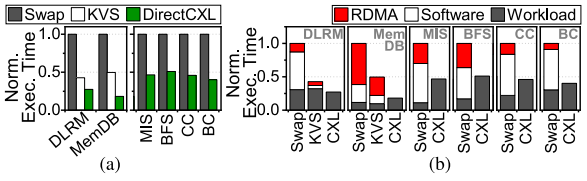

FIGURE 8. Real workload performance. (a) Execution time. (b) Execution breakdown.

`RDMA` when the size of payloads is smaller than 1 KB (4,158 cycles, on average). As the payloads increase, `Copy` gets longer and reaches 28.9% of total execution time. This is because users must copy all their data into RNIC's MR, which takes extra overhead in `RDMA`. On the other hand, `Memory` and `Network` shows a performance trend similar to `RDMA` analyzed in Figure 6(a). Note that the actual times of `Network` [see Figure 6(b)] do not decrease as the payload increases; while `Memory` increases to handle large size of data, RNIC can simultaneously transmit the data to the underlying network. These overlapped cycles are counted by `Memory` in our analysis. As shown in Figure 6(c), the breakdown analysis for `DirectCXL` shows a completely different story; there is neither software nor data copy overhead. As the payloads increase, the dominant component of `DirectCXL`'s latency is LLC (`CPU Cache`). This is because LLC can handle 16 concurrent misses through miss status holding registers in our custom CPU. Thus, many memory requests (64 bytes) composing a large payload data can be stalled at CPU, which takes 67% of the total latency to handle 4-KB payloads. `PCIe` shown in Figure 6(b) does not decrease as the payloads increase because of a similar reason of `RDMA`'s `Network`. However, it is not as much as what `Network` did as only 16 concurrent misses can be overlapped. Note that `PCIe` shown in Figure 6(a) and (c) includes the latency of CXL IPs (RP, EP, and switch), which is different from the pure cycles of PCIe physical bus. The pure cycles of PCIe physical bus (FlexBus) account for 28% of `DirectCXL` latency.

Memory hierarchy performance: Figure 7 shows latency cycles of different components in the system's memory hierarchy. While `Local` and `DirectCXL` exhibits `CPU cache` by lowering the memory access latency to four cycles, `RDMA` has negligible impacts on `CPU cache` as their network overhead is much higher than that of `Local`. The best-case performance of `RDMA` was 2,027 cycles, which is 6.2× and 510.5× slower than that of `DirectCXL` and L1 cache, respectively. `DirectCXL` requires 328 cycles whereas `Local` requires only 60 cycles in the case of L2 misses. Note that the performance bottleneck of `DirectCXL` is `PCIe` including CXL IPs (77.8% of the total latency).

Performance of Real Workloads

Figure 8(a) shows the execution latency of `Swap`, `KVS`, and `DirectCXL` when running DLRM, MemDB, and four workloads from Ligra. For better understanding, all the results in this section are normalized to those of `Swap`. For Ligra, we only compare `DirectCXL` with `Swap` because Ligra's graph processing engines (handling in/out edges and vertices) is not compatible with a key-value structure. `KVS` can reduce the latency of `Swap` as it addresses the overhead imposed by page-based I/O granularity to access the remote memory. However, it has two major issues behind `KVS`. First, it requires significant modification of the application's source codes, which is often unable to service (e.g., MIS, BFS, CC, and BC). Second, `KVS` requires heavy computation, such as hashing at the memory node, which increases monetary costs. In contrast, `DirectCXL` without having a source modification and remote-side resource exhibits 3× and 2.2× better performance than `Swap` and even `KVS`, respectively.

To better understand this performance improvement of `DirectCXL`, we also decompose the execution times into `RDMA`, network library intervention (`Software`), and application execution itself (`Workload`), and the results are shown in Figure 8(b). This figure demonstrates where `Swap` degrades the overall performance from its execution; 51.8% of the execution time is consumed by kernel swap daemon and FastSwap driver, on average. This is because `Swap` just expands memory with the local and remote based on LRU, which makes its page exchange frequent. The reason why `KVS` shows performance better than `Swap` in the

TABLE 2. Graph dataset characteristics.

Graph datasets	Sampled vertices	Sampled embeddings
ogbn-arxiv	27,822	13.6 MB
citeseer	1,231	17.4 MB
ogbn-mag	39,087	19.1 MB
cora_ml	2,107	23.1 MB
dblp	7,261	45.4 MB
yelp	59,988	68.7 MB
cora	7,886	262.0 MB
cs	11,064	287.2 MB
physics	19,144	614.5 MB

cases of DLRM and MemDB is mainly related to workload characteristics and its service optimization. For DLRM, KVS loads the exact size of embeddings rather than a page, which reduces Swap's data transfer overhead as high as 6.9×. While KVS shows the low overhead in our evaluation, RDMA and Software can linearly increase as the number of inferences increases; in our case, we only used 13.5 MB (0.0008%) of embeddings for single inference. For MemDB, as KVS stores all key-value pairs into local DRAM, it only accesses remote-side DRAM to inquiry values. However, it spends 55.3% and 24.9% of the execution time for RDMA and Software to handle the remote DRAMs, respectively. In contrast, DirectCXL removes such hardware and software overhead, which exhibits much better performance than Swap and KVS. Finally, all the four graph workloads show similar trends; Swap is always slower than DirectCXL. They require multiple graph traverses, which frequently generate random memory access patterns. As Swap requires exchanging 4-KB pages to read 8-byte pointers for graph traversing, it shows 2.2× worse performance than DirectCXL.

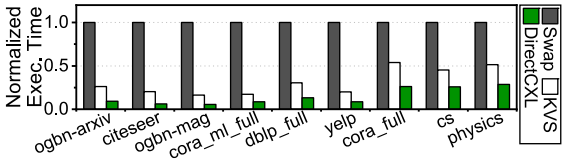

FIGURE 9. Performance of embedding lookup.

Geometrical Deep Learning

Graph neural networks (GNNs)[9] can also take the benefits from a large storage capacity that memory pooling offers. In this section, we evaluate three inference systems, each using Swap, KVS, and DirectCXL, respectively. All inferences of the systems are performed through a graph convolutional network[10] with a popular graph sampling technique.[11] We use the nine real-world graphs for the inference systems. In this evaluation, the number of sampled vertices used for inferences range from 1 K to about 59 K, and the corresponding embedding sizes are analyzed in Table 2. This information is served from all the remote memory nodes.

Figure 9 shows the embedding lookup latency of each target graph workloads; the latency values of KVS and DirectCXL are normalized to those of Swap for better understanding. KVS exhibits 3.9× better performance than Swap, on average. The main reason behind this shorter latency is KVS's efficient use of the system's local memory. Note that Swap has no policy to determine which data should be located to its local memory and which data should be in the remote side. As the local memory is insufficient to accommodate all embeddings and the runtime data, the embeddings are often swapped out after graph sampling and inference processing. In contrast, KVS explicitly places all the embeddings to the remote memory nodes, and thus, it does not have such data movement imposed by the capacity limit of the local memory. DirectCXL can take all the advantages mentioned previously as it

TABLE 3. Feature comparison of different CXL versions.

Features	CXL 1.1[12]	CXL 2.0[13]	CXL 3.0 (Ongoing work)	Integration
New opcodes/fields	✘	✘	✔ 256B Flit support	Link Layer
	✘	✘	✔ 16 devices per RP	CXL.cache
Multiple CXL.cache devices	✘	✘	✔ CXL.cache Routing	Switch
Back invalidation	✘	✘	✔ Back invalidation	CXL.mem
	✘	✔	✔ Multilogical device	Trans. Layer
Fine-control memory sharing	✘	✘	✔ Multiheaded device	Trans. Layer
	✘	✘	✔ Dynamic Capacity	CXL.mem/SW
	✘	✘	✔ Port-based Routing	Switch
Fabric extension	✘	✘	✔ Multi-level switch	Switch/SW
	✘	✘	✔ Fabric-attached memory	Switch/SW

also stores the embedding information to the remote memory. However, DirectCXL does not require software intervention and overhead associated with RNIC control and the corresponding data management, which makes DirectCXL 2.4× better than KVS, on average.

TOWARD COMPOSABLE SERVER ARCHITECTURE

In this section, we introduce our on-going and future work. While DirectCXL is designed toward scalable memory pooling, there are several limits stemming from CXL 2.0. For better understanding, we compare each version of existing CXL protocols and summarize them in Table 3. The table also includes the specific information about where our on-going work integrates new features (enhanced by CXL 3.0) into (e.g., link/transaction layers and software).

Multiple cache devices: The most important features that our on-going work integrate is 1) multiple cache devices support; 2) back-invalidation snoop; and 3) memory sharing and fine control capability. As shown in Figure 10(a), our CXL 3.0-based on-going prototype allows a RP to have sixteen type-1 and/or type-2 devices (e.g., domain specific or ML accelerators) at most, which was prohibited in CXL 2.0. Thus, a different type of accelerators can be placed into a same cache coherent domain per host, making heterogenous computing much more scalable. To this end, CXL 3.0 extents the CXL flit (66 bytes) to a 256-byte flit and add a set of new fields, including cache identifiers (4 bits) with other information, which will be explained shortly. Note that this information was not necessary in DirectCXL as CXL 2.0 only allows a CXL RP to have a single type-1 or type-2 device.

Back-invalidation snoop: Further, our CXL 3.0-based on-going work places type-3 devices (memory expanders) in the CXL's cache coherent domain by introducing a new operation code (opcode) to the 256 bytes flit, called back invalidation. Each type-3 device can invalidate the host-side processor cache through back invalidation, which allows the underlying devices to control all cache states of the corresponding host processor (s) or type-1/2 devices in an active manner. The back-invalidation snoop thus makes all types of CXL devices coherent, which can tightly integrate computing and memory domains into a single cache coherent domain.

Memory expansion control: CXL 3.0 provides a set of new features allowing the underlying memory expanders (type-3) more manageable and sharable across different host processors and accelerators in a fine granular manner. The ongoing platform specifically utilizes multiheaded logical device (MH-LD) and dynamic capacity device (DCD) feature, each being implemented in CXL's transaction layer and system software, respectively. Figure 10(b) analyzes the difference between DirectCXL and the CXL 3.0-based on-going prototype from those two features. DirectCXL (shown in the figure's right) can partition a physical type-3 device to multiple logical devices. While each logical device can be mapped to different hosts through different HDMs, it mainly has two technical issues making CXL 2.0 difficult to be in a large scale: 1) static partition with low bandwidth and 2) exclusive memory binding. The memory bandwidth is significantly limited because each HDM is exposed to one or more hosts via the device's single PCIe path (called *head*). In contrast, our on-going work utilizes MH-LD (CXL 3.0) allows the type-3 device to have multiple heads, each being able to be individually partitioned into multiple logical devices [cf., left-hand side of Figure 10(b)]. As there are multiple heads, HDMs can be exposed to the host without bandwidth degradation. In contrast to CXL 2.0, each HDM can be allocated to (and shared with) different hosts through DCD, which will be explained shortly.

Data sharing and dynamic capacity support: Our on-going project leverages CXL 3.0's new feature, DCD for better memory utilization. In contrast to CXL

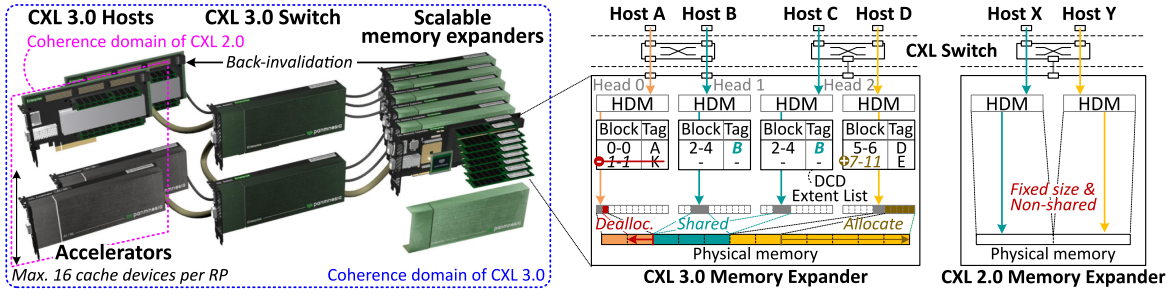

FIGURE 10. Composable CXL server architecture. (a) Example topology and coherence domain comparison. (b) Fine-control memory sharing of CXL 3.0.

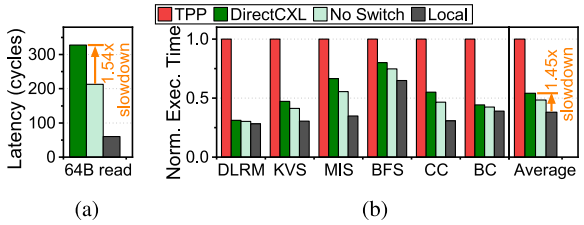

FIGURE 11. Performance comparisons. (a) Memory latency. (b) Latency of real workloads.

2.0, it manages HDM's physical memory space using many of small sized data chunks, called blocks. The blocks (associated with the corresponding HDM's host) can be dynamically reallocated to other host processors or accelerators. To this end, as shown in Figure 10(b), it employs a DCD extent list, containing the block and tag information. What the tag (10 bytes) can maintain is not well defined in CXL 3.0, in our on-going project; it includes host identifiers and be used to distinguish which host owns the corresponding block. Thus, HDM can allocate available blocks residing in anywhere of the DCD extent list such that our on-going work can meet the load balancing requirement coming from diverse hosts. Further, it can support data sharing with assist of the back-invalidation snoop.

Fabric extension: Since CXL 2.0 leverages most features of PCIe, it uses 8-bit unique device identifiers (bus) for the underlying device discovery and enumeration. This limits the number of devices that can be in a CXL interconnect (as many as 253 in theory). Our on-going work uses CXL 3.0's fabric style topology, which uses port numbers for the devices instead of device identifiers. To this end, our CXL switch employs a port-based routing (PBR). It basically extents CXL flits to PBR flits and routes the flits based on specified port numbers. This allows CXL to integrates more CXL switches into a coherence domain, called multilevel switch in CXL 3.0 thereby having upto 4,096 devices and/or hosts. By putting all the features mentioned previously together (cf., Table 3), this CXL fabric extension is expected to be heterogenous, sharable, and highly scalable.

DISCUSSION

Related work: Very recently, two CXL-based studies have been updated in an open-access archive.[12,14] Pond[14] utilizes CXL-attached memory to handle untouched memory spaces, allocated by a set of user-level applications running on a virtual machine. It employs a machine learning technique to determine which portion of memory can be offloaded to the CXL-attached memory without performance degradation. DirectCXL also can be extended to virtual machine, by modifying hypervisor to allocate memory from the CXL-attached memory. TPP[12] leverages NUMA balancing to use CXL-attached memory as a second memory tier. As NUMA balancing is based on minor page faults, it requires software intervention for proper operation. In contrast, DirectCXL makes user-level application to use CXL-attached memory without any software intervention.

Performance comparison: To better understand the performance trends of CXL, we compare our DirectCXL with a local memory only system (`Local`), `DirectCXL` without the CXL switch (`No Switch`), and TPP (`TPP`). We run same benchmarks, which are used for the memory hierarchy test and the real-world workload tests in the "Evaluation" section.

Figure 11(a) shows the result of the memory hierarchy test. While the switch is essential to realize flexible memory pooling, switch introduces additional latency. As shown in figure, `DirectCXL` is 1.54× slower than `No Switch`, because of flit routing latency and the serialization/deserialization at the CXL IPs.

> *THE RESULTS OF OUR END-TO-END REAL SYSTEM EVALUATION SHOW THAT THE DISAGGREGATED MEMORY RESOURCES OF DirectCXL CAN EXHIBIT DRAM-LIKE PERFORMANCE WHEN THE WORKLOAD CAN ENJOY THE HOST-PROCESSOR'S CACHE.*

Figure 11(b) shows the normalized execution time of the real-world workloads. While `DirectCXL` is 5.5× slower than `Local` in the memory hierarchy test, `DirectCXL` is 1.45× slower than `Local`, on average, in the real-world workload tests. This performance difference is originated from memory access pattern and prefetching. If application has locality on memory accesses, CPU cache can serve data without accessing the CXL-attached memory. In addition, our RISC-V CPU has hardware prefetcher in L1D cache, effectively hides long CXL latency on sequential memory accesses. For `No Switch`, it shows 10.2% better performance than `DirectCXL`, on average, with a penalty of nonviable memory pooling. For `TPP`, we emulated it using FastSwap without network access, because its source code is not released. Based on the emulation, `TPP` will have 2.03× lower performance than DirectCXL, on average.

CONCLUSION

We proposed DirectCXL that connects host processor complex and remote memory resources over CXL's memory protocol (CXL.mem). The results of our real system evaluation show that the disaggregated memory resources of DirectCXL can exhibit DRAM-like performance when the workload can enjoy the host-processor's cache. For real-world applications, it exhibits 7× better performance than RDMA-based memory disaggregation, on average.

ACKNOWLEDGMENTS

This article is a full-length version of an earlier six-page conference paper.[13] We appreciate anonymous reviewers, and thank Panmnesia for all the technical support. This work is protected by one or more patents.

REFERENCES

1. E. Amaro et al., "Can far memory improve job throughput?," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.
2. K. Lim et al., "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, 2012, pp. 1–12.
3. C. Pinto et al., "ThymesisFlow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation," in *Proc. IEEE 53rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 868–880.
4. Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "CLIO: A hardware-software co-designed disaggregated memory system," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022, pp. 417–433.
5. A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 295–306.
6. Z. Ruan,M. M. K. S. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 315–332.
7. P. W. Frey and G. Alonso, "Minimizing the hidden cost of RDMA," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2009, pp. 553–560.
8. J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.
9. F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.
10. M. Welling and T. N. Kipf, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–11. [Online]. Available: https://openreview.net/pdf?id=SJU4ayYgl
11. W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Neural Inf. Process. Syst.*, 2017, pp. 1–11. [Online]. Available: https://papers.nips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf
12. H. A. Maruf, et al., "TPP: Transparent page placement for CXL-enabled tiered memory," 2022, *arXiv:2206.02878*.
13. D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, high-performance memory disaggregation with DirectCXL," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 287–294.
14. H. Li et al., "Pond: CXL-Based memory pooling systems for cloud platforms," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2023.

DONGHYUN GOUK is currently working toward a Ph.D. degree at KAIST, Daejeon, 34141, South Korea. His research interests include computing interconnects, ML accelerators, RISC-V, and CXL.

MIRYEONG KWON is currently working toward a Ph.D. degree at KAIST, Daejeon, 34141, South Korea. Her research interests include graph analysis, GPU/SSD integration, persistent memory, and CXL.

HANYEOREUM BAE is currently working toward a Ph.D. degree at KAIST, Daejeon, 34141, South Korea. His research interests include new storage, ZNS, IOD, and CXL.

SANGWON LEE is currently working toward a Ph.D. degree at KAIST, Daejeon, 34141, South Korea, and also works at Panmnesia, Inc. His research interests include persistent memory, RISC-V, and CXL.

MYOUNGSOO JUNG is an associate professor in the School of Electrical Engineering, KAIST, Daejeon, 34141, South Korea, since 2019. He is also the CEO of Panmnesia, Inc, Daejeon, South Korea. His research interests include computer architecture, operating system, storage systems, nonvolatile memory, parallel processing, heterogeneous computing, and CXL. He is a corresponding author of this article. Contact him at mj@camelab.org.