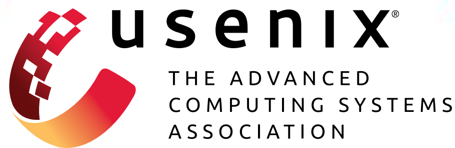


Direct Access, High-Performance Memory Disaggregation with DirectCXL

Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung, ***Computer Architecture and Memory Systems Laboratory, Korea Advanced Institute of Science and Technology (KAIST)***

https://www.usenix.org/conference/atc22/presentation/gouk

This paper is included in the Proceedings of the
2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the
2022 USENIX Annual Technical Conference
is sponsored by

Direct Access, High-Performance Memory Disaggregation with DirectCXL

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

Abstract

New cache coherent interconnects such as CXL have recently attracted great attention thanks to their excellent hardware heterogeneity management and resource disaggregation capabilities. Even though there is yet no real product or platform integrating CXL into memory disaggregation, it is expected to make memory resources practically and efficiently disaggregated much better than ever before.

In this paper, 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`). To this end, we explore a practical design for CXL-based memory disaggregation and make it real. 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 via sheer load/store instructions. Since DirectCXL does not require any data copies between the host memory and remote memory, it can expose the true performance of remote-side disaggregated memory resources to the users.

1 Introduction

Memory disaggregation has attracted great attention thanks to its high memory utilization, transparent elasticity, and resource management efficiency [1–3]. Many studies have explored various software and hardware approaches to realize memory disaggregation and put significant efforts into making it practical in large-scale systems [4–16].

We can broadly classify the existing memory disaggregation runtimes into two different approaches based on how they manage data between a host and memory server(s): i) page-based and ii) object-based. The page-based approach [4–10] utilizes virtual memory techniques to use disaggregated memory without a code change. It swaps page cache data residing on the host's local 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 instead of leveraging the virtual memory systems [11–16]. 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*) [4, 5, 11–13, 15, 16] (or similar fine-grain network interfaces [7, 9, 10, 17]). 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 [18]), which is a new concept of open industry standard interconnects offering high-performance connectivity among multiple host processors, hardware accelerators, and I/O devices [19]. 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 over-provisioning with low latency [20–22]. Even though CXL exhibits a great potential to realize memory disaggregation 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 disaggregation 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 that employs multiple DRAM modules on a remote side. We then prototype a set of network infrastructure components such as a CXL switch in order to make the disaggregated memory connected to the host in a scalable manner. As there is no operating system that support 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 prototype DirectCXL using many cus-

tomized memory add-in-cards, 16*nm* FPGA-based processor nodes, a switch, and a PCIe backplane. On the other hand, DIRECTCXL software runtime is implemented based on Linux 5.13. To the best of our knowledge, this is the first work that brings CXL 2.0 into a real system and analyzes the performance characteristics of CXL-enabled disaggregated memory design. 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 3× better performance than RDMA-based memory disaggregation, on average.

2 Memory Disaggregation and Related Work

2.1 Remote Direct Memory Access

The basic idea of memory disaggregation is to connect a host with one or more memory nodes, such that it does not restrict a given job execution because of limited local memory space. For the backend network control, most disaggregation work employ remote direct memory access (RDMA) [4, 5, 11–13, 15, 16] or similar customized DMA protocols [7, 9, 10]. 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* [23]), 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 [24].

2.2 Swap: Page-based Memory Pool

Page-based memory disaggregation [4–10] 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 [16].

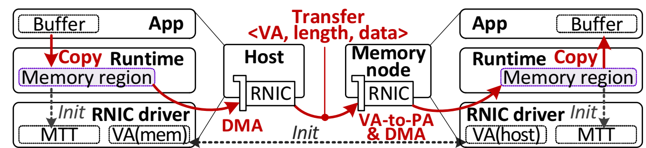


Figure 1: Data movement over RDMA.

Note that there are several studies that migrate locally cached data in a finer granular manner [4–7] or reduce the page fault overhead by offloading memory management (including page cache coherence) to the network [8] or memory nodes [9, 10]. However, all these approaches use RDMA (or a similar network protocol), which is essential to cache the data and pay the cost of memory operations for network handling.

2.3 KVS: Object-based Memory Pool

In contrast, object-based memory disaggregation systems [11–16] directly intervene in RDMA data transfers using their own database such as key-value store (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 hash-table 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.

3 Direct Accessible Memory Aggregation

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.

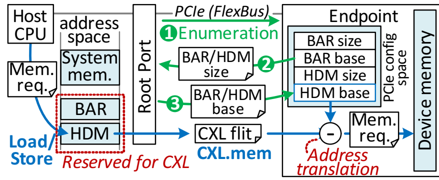
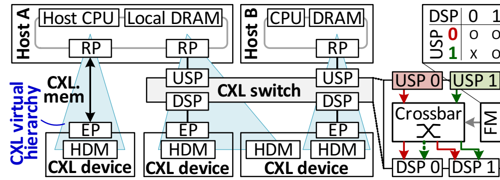


Figure 2: DirectCXL's connection method.



(a) CXL virtual hierarchy. (b) CXL switch.

Figure 3: DirectCXL's network and switch.

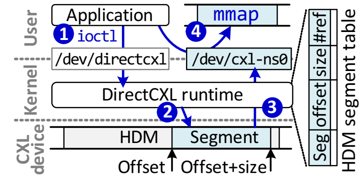


Figure 4: DirectCXL software runtime.

3.1 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 DIMMs with its own hardware controllers. Our CXL device employs multiple DRAM controllers, connecting DRAM DIMMs over the conventional 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.

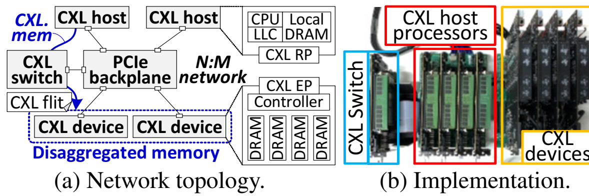
Integrating devices into system memory. Figure 2 shows how CXL devices' internal DRAMs are mapped (exposed) to a host's memory space over CXL. The host CPU's system bus contains one or more CXL root ports (*RPs*), which connect one or more CXL devices as endpoint (*EP*) devices. Our host-side kernel 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. 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 3a illustrates how DirectCXL can disaggregate memory resources from a host using one or more and CXL devices, and Figure 3b 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.

3.2 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 multiple 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 `/dev/directcxl`, 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



(a) Network topology. (b) Implementation.

Figure 5: CXL-enabled cluster.

application can then map the cxl-namespace to its process virtual memory space using `mmap` with `vm_area_struct`.

Note that DIRECTCXL software runtime is designed for direct access of CXL devices, which is a similar concept to the memory-mapped file management of persistent memory development toolkit (PMDK [25]). However, it is much simpler and more flexible for namespace management than PMDK. For example, PMDK's namespace is very much the same idea as NVMe namespace, managed by file systems or DAX with a fixed size [26]. In contrast, our cxl-namespace is more similar to the conventional memory segment, which is directly exposed to the application without a file system employment.

3.3 Prototype Implementation

Figure 5a illustrates our design of a CXL network topology to disaggregate memory resources, and the corresponding implementation in a real system is shown in Figure 5b. There are n numbers of compute hosts connected to m number of CXL devices through a CXL switch; in our prototype, n and m are four, but those numbers can scale by having more CXL switches. Specifically, each CXL device prototype is built on our customized add-in-card (AIC) CXL memory blade that employs 16*nm* FPGA and 8 different DDR4 DRAM modules (64GB). 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 four CXL devices (32 DRAM modules) to the four 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. The host-side processors require advanced configuration and power interface (ACPI [27]) for CXL 2.0 enumeration (e.g., RP location and RP's reserved address space). Since RISC-V does not support ACPI yet, we enable the CXL enumeration by adding such information into the device tree [28]. Specifically, we update an MMIO register designated as a property of the tree's node to let the processor know where CXL RP exists. On the other hand, we add a new field (`cxl-reserved-area`) in the node to indicate where an HDM can be mapped. Our in-house softcore processors work at 100MHz while CXL and PCIe IPs (RP, EP, and Switch) operate at 250MHz.

4 Evaluation

Testbed prototypes for memory disaggregation. In addition to the CXL environment that we implemented in Section 3.3 (`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 (56Gbps, [29]) instead of our CXL switch as RDMA network interface card (RNIC). In addition, we port Mellanox OpenFabric Enterprise Distribution (OFED) v4.9 [30] as an RDMA driver to enable RNIC in our evaluation testbed. Lastly, we port FastSwap [1] and HERD [12] 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 above 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 line-up.

Benchmark and workloads. Since there is no microbenchmark that we can compare different memory pooling technologies (`RDMA` vs. `DirectCXL`), we also build an in-house memory benchmark for in-depth analysis of those two technologies (Section 4.1). 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 pre-allocated 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 [31]), an in-memory database used for the HERD evaluation (MemDB [12]), and four graph analysis workloads (MIS [32], BFS [33], CC [34], and BC [35]) coming from Ligra [36]. All their tables and data structures are stored in the remote node, while each host's local memory handles the execution code and static data. Table 1 summarizes the per-node memory usage and total data sizes for each workload that we tested.

4.1 In-depth Analysis of RDMA and CXL

In this subsection, we compare the performance of RDMA and CXL technologies when the host and memory nodes are

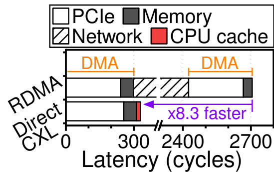
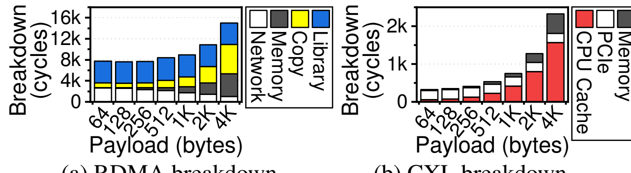


Figure 6: RDMA vs. CXL.



(a) RDMA breakdown. (b) CXL breakdown.

Figure 7: Sensitivity tests.

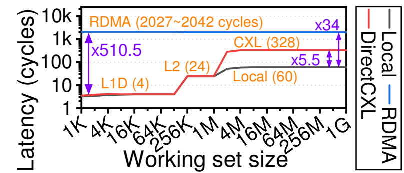


Figure 8: Memory hierarchy performance.

configured through a 1:1 connection. Figure 6 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% (2129 cycles) of the total latency (2705 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 7a 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 `RDMA` when the size of payloads is smaller than 1KB (4158 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. Note that the actual times of `Network` (Figure 7a) 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 7b, 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 (MSHR) in our custom CPU. Thus, many memory requests (64B) composing a large payload data can be stalled at CPU, which takes 67% of the total latency to handle 4KB payloads. `PCIe` shown in Figure 7a 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. ote that `PCIe` shown in Figures 6 and 7b 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. The detailed latency decomposition will be analyzed in Section 4.2.

	Per-node usage		Total usage	Data stored in remote memory
	Local	Remote		
DLRM [31]	Less than 100MB	17GB	68GB	Embedding tables.
MemDB [12]		4GB	16GB	Key-value pairs and tree structure.
Ligra [36]		7GB	28GB	Deserialized graph structure.

Table 1: Memory usage characteristic of each workload.

Memory hierarchy performance. Figure 8 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 4 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 2027 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). This can be accelerated by increasing the working frequency, which will be discussed shortly.

4.2 Latency Distribution and Scaling Study

Latency distribution. In addition to the latency trend (average) we reported above, we also analyze complete latency behaviors of `Local`, `RDMA`, and `DirectCXL`. Figure 9 shows the latency CDF of memory accesses (64B) for the different pooling methods. `RDMA` shows the performance curve, which ranges from 1790 cycles to 4006 cycles. The reason why there is a difference between the minimum and maximum latency of `RDMA` is RNIC's MTT memory buffer and CPU caches for data transfers. While `RDMA` cannot take the benefits from direct load/store instruction with CPU caches, its data transfers themselves utilize CPU caches. Nevertheless, `RDMA` cannot avoid the network accesses for remote memory accesses, making its latency worse than `Local` by 36.8×, on average. In contrast, the latency behaviors of `DirectCXL` are similar to `Local`. Even though the latency of `DirectCXL` (reported in Figures 6 and 7b) is the average value, its best performance is the same as `Local` (4∼24 cycles). This is because, as we showed in the previous section, `DirectCXL` can take the benefits of CPU caches directly. The tail latency is 2.8× worse than `Local`, but its latency curve is similar to that of `Local`. This is because both `DirectCXL` and `Local` use the same DRAM (and there is no network access overhead).

Speed scaling estimation. The cycle numbers that we reported here are measured at each host's CPU using register-level instrumentation. We believe it is sufficient and better

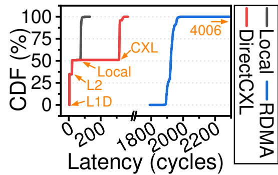
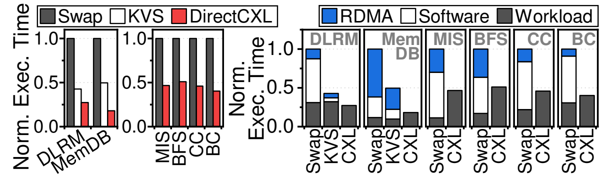


Figure 9: Memory-level latency CDF (64B).

Measurement clock domain →		DirectCXL CPU (100MHz)	PCIe 5.0 x8 (Estimated) CPU (1.2GHz)	Time delay
L1/L2 cache		30	30	25 ns
PCIe	CXL IPs (2.0)*	165	287	239 ns
	FlexBus	91	69	57 ns
DRAM controller		42	126	105 ns
Total		328	512	426 ns

*Including RP, EP, and Switch. Unit: cycles

Table 2: Latency breakdown and estimated 64B load latency.



(a) Execution Time. (b) Execution breakdown.

Figure 10: Real workload performance.

than a cross-time-domain analysis to decompose the system latency. Nevertheless, we estimate a time delay in cases where the target system accelerates the frequency of its processor complex and CXL IPs (RP, EP, and Switch) by 1.2GHz and 1GHz, respectively. Table 2 decomposes `DirectCXL`'s latency of a 64B memory load and compares it with the estimated time delay. The cycle counts of L1/L2 cache misses are not different as they work in all the same clock domain of CPU. While other components (FlexBus, CXL IPs, and DRAM controller) speed up by 4× (250MHz → 1GHz), the number of cycles increases since CPU gets faster by 12×. Note that, as the version of PCIe is changed and the number of lanes for PCIe increases by double, FlexBus's cycles decrease. The table includes the time delays corresponding to the estimated system from the CPU's viewpoint. While the time delay of FlexBus is pretty good (∼60ns), the corresponding CXL IPs have room to improve further with a higher working frequency.

4.3 Performance of Real Workloads

Figure 10a 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 subsection 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, 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`) latencies, and the results are shown in Figure 10b. This figure demonstrates where `Swap` degrades the overall performance from its execution; 51.8% of the execution time is consumed by kernel swap daemon (`kswapd`) 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 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 13.5MB (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`. Note that MemDB contains 2M key-value pairs whose value size is 2KB, and its host queries 8M `Get` requests by randomly generating their keys. This workload characteristic roughly makes `DirectCXL`'s memory accesses be faced with a cache miss for every four queries. Note that `Workload` of `DirectCXL` is longer than that of `KVS`, because `DirectCXL` places all hash table and tree for key-value pairs whereas `KVS` has it in local DRAM. Lastly, 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 4KB pages to read 8B pointers for graph traversing, it shows 2.2× worse performance than `DirectCXL`.

5 Conclusion

In this paper, we propose 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 3× better performance than RDMA-based memory disaggregation, on average.

6 Future Work and Acknowledgement

The authors are extending the kernel for efficient CXL memory management and consider having an SoC silicon as future work of `DirectCXL`. This work is protected by one or more patents. The authors would like to thank the anonymous reviewers for their comments, and Myoungsoo Jung is the corresponding author (mj@camelab.org).

References

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[2] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. Memory disaggregation: Research problems and opportunities. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1664–1673. IEEE, 2019.

[3] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[4] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[5] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018.

[6] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[7] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. Thymesisflow: a software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880. IEEE, 2020.

[8] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[9] 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*, pages 417–433, 2022.

[10] 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*, pages 79–92, 2021.

[11] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48, 2020.

[12] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70, 2015.

[15] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[16] 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 20)*, pages 315–332, 2020.

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

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

[19] CXL Consortium. Compute Express Link™ 2.0 White Paper. https://www.computeexpresslink.org/_files/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf.

[20] Navin Shenoy. A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data.

[21] Debendra Das Sharma. CXL: Coherency, Memory, and I/O Semantics on PCIe Infrastructure. https://www.electronicdesign.com/technologies/embedded-revolution/article/21162617/cxl-coherency-memory-and-io-semantics-on-pcie-infrastructure.

[22] Patrick Kennedy. Compute Express Link or CXL What it is and Examples. https://www.servethehome.com/compute-express-link-or-cxl-what-it-is-and-examples/.

[23] Hari Subramoni, Ping Lai, Miao Luo, and Dhabaleswar K Panda. Rdma over ethernet—a preliminary study. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9. IEEE, 2009.

[24] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009.

[25] Intel. Persistent Memory Developer Kit Version v1.11.0. https://pmem.io/.

[26] Intel. NVDIMM Namespace Specification. https://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

[27] UEFI Forum, Inc. Advanced Configuration and Power Interface (ACPI) Specification Version 6.4. https://uefi.org/specs/ACPI/6.4/, 2021.

[28] Linaro. The devicetree specification. https://www.devicetree.org/.

[29] Mellanox. Mellanox ConnectX-3 FDR (56Gbps) InfiniBand VPI. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX3_VPI_Card_Dell.pdf.

[30] Xilinx. Mellanox OpenFabrics Enterprise Distribution. https://www.mellanox.com/products/infiniband-drivers/linux/mlnx_ofed.

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[32] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[33] Alan Bundy and Lincoln Wallen. Breadth-first search. In *Catalogue of artificial intelligence tools*, pages 13–13. Springer, 1984.

[34] Fan Chung and Linyuan Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.

[35] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[36] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.