Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices

Yan Sun
University of Illinois
Urbana, U.S.A.
yans3@illinois.edu

Yifan Yuan
Intel Labs
Hillsboro, U.S.A.
yifan.yuan@intel.com

Zeduo Yu
University of Illinois
Urbana, U.S.A.
zeduoyu2@illinois.edu

Reese Kuper
University of Illinois
Urbana, U.S.A.
rkuper2@illinois.edu

Chihun Song
University of Illinois
Urbana, U.S.A.
chihuns2@illinois.edu

Jinghan Huang
University of Illinois
Urbana, U.S.A.
jinghan4@illinois.edu

Houxiang Ji
University of Illinois
Urbana, U.S.A.
hj14@illinois.edu

Siddharth Agarwal
University of Illinois
Urbana, U.S.A.
sa10@illinois.edu

Jiaqi Lou
University of Illinois
Urbana, U.S.A.
jiaqil6@illinois.edu

Ipoom Jeong
University of Illinois
Urbana, U.S.A.
ipoom@illinois.edu

Ren Wang
Intel Labs
Hillsboro, U.S.A.
ren.wang@intel.com

Jung Ho Ahn
Seoul National University
Seoul, Republic of Korea
gajh@snu.ac.kr

Tianyin Xu
University of Illinois
Urbana, U.S.A.
tyxu@illinois.edu

Nam Sung Kim
University of Illinois
Urbana, U.S.A.
nskim@illinois.edu

ABSTRACT

The ever-growing demands for memory with larger capacity and higher bandwidth have driven recent innovations on memory expansion and disaggregation technologies based on Compute eXpress Link (CXL). Especially, CXL-based memory expansion technology has recently gained notable attention for its ability not only to economically expand memory capacity and bandwidth but also to decouple memory technologies from a specific memory interface of the CPU. However, since CXL memory devices have not been widely available, they have been emulated using DDR memory in a remote NUMA node. In this paper, for the first time, we comprehensively evaluate a true CXL-ready system based on the latest 4$^{\text{th}}$-generation Intel Xeon CPU with three CXL memory devices from different manufacturers. Specifically, we run a set of microbenchmarks not only to compare the performance of true CXL memory with that of emulated CXL memory but also to analyze the complex interplay between the CPU and CXL memory in depth. This reveals important differences between emulated CXL memory and true CXL memory, some of which will compel researchers to revisit the analyses and proposals from recent work. Next, we identify opportunities for memory-bandwidth-intensive applications to benefit from the use of CXL memory. Lastly, we propose a CXL-memory-aware dynamic page allocation policy, `Caption` to more efficiently use CXL memory as a bandwidth expander. We demonstrate that `Caption` can automatically converge to an empirically favorable percentage of pages allocated to CXL memory, which improves the performance of memory-bandwidth-intensive applications by up to 24% when compared to the default page allocation policy designed for traditional NUMA systems.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**; • **Computer systems organization** → **Architectures**; • **General and reference** → **Measurement**.

KEYWORDS

Compute eXpress Link, tiered-memory management, measurement

ACM Reference Format:
Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3613424.3614256

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614256

1 INTRODUCTION

Emerging applications have demanded memory with even larger capacity and higher bandwidth at lower power consumption. However, as the current memory technologies have almost reached their scaling limits, it has become more challenging to meet these demands cost-efficiently. Especially, when focusing on the memory

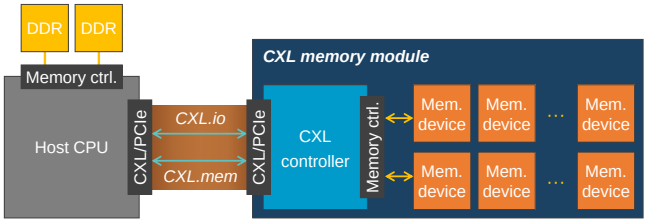


Figure 1: CXL memory module architecture.

interface technology, we observe that DDR5 requires 288 pins per channel [44], making it more expensive to increase the number of channels for higher bandwidth under the CPU's package pin constraint. Besides, various signaling challenges in high-speed parallel interfaces, such as DDR, make it harder to further increase the rate of data transfers. This results in super-linearly increasing energy consumption per bit transfer [55] and reducing the number of memory modules (DIMMs) per channel to one for the maximum bandwidth [33]. As the capacity and bandwidth of memory are functions of the number of channels per CPU package, the number of DIMMs per channel, and the rate of bit transfers per channel, DDR has already shown its limited bandwidth and capacity scalability. This calls for alternative memory interface technologies and memory subsystem architectures.

Among them, Compute eXpress Link (CXL) [73] has emerged as one of the most promising memory interface technologies. CXL is an open standard developed through a joint effort by major hardware manufacturers and hyperscalers. As CXL is built on the standard PCIe, which is a serial interface technology, it can offer much higher bit-transfer rates per pin (*e.g.*, PCIe 4.0: 16 Gbps/lane vs. DDR4-3200: 3.2 Gbps/pin) and consumes lower energy per bit transfer (*e.g.*, PCIe 4.0: 6 pJ/bit [75] vs. DDR4: 22 pJ/bit [56]), but at the cost of much longer link latency (*e.g.*, PCIe 4.0: ~40 ns [73] vs. DDR4: <1 ns [15]). Compared to PCIe, CXL implements additional features that enable the CPU to communicate with devices and their attached memory in a cache-coherent fashion using load and store instructions. Figure 1 illustrates a CXL memory device consisting of a CXL controller and memory devices. Consuming ~3× fewer pins than DDR5, a CXL memory device based on PCIe 5.0 ×8 may expand memory capacity and bandwidth of systems cost-efficiently. Furthermore, with the CXL controller between the CPU and memory devices, CXL decouples memory technologies from a specific memory interface technology supported by the CPU. This grants memory manufacturers unprecedented flexibility in designing and optimizing their memory devices. Besides, by employing retimers and switches, a CPU with CXL support can easily access memory in remote nodes with lower latency than traditional network interface technologies like RDMA, efficiently facilitating memory disaggregation. These advantages position memory-related extension as one of the primary target use cases for CXL [25, 32, 65, 67], and major hardware manufacturers have announced CXL support in their product roadmaps [4, 25, 35, 65, 67].

Given its promising vision, CXL memory has recently attracted significant attention with active investigation for datacenter-scale deployment [59, 64]. Unfortunately, due to the lack of commercially available hardware with CXL support, most of the recent research on CXL memory has been based on emulation using memory in a remote NUMA node in a multi-socket system, since CXL memory is exposed as such [6, 59, 64]. However, as we will reveal in this paper, there are fundamental differences between emulated CXL memory and true CXL memory. That is, the common emulation-based practice of using a remote NUMA node to explore CXL memory may give us misleading performance characterization results and/or lead to suboptimal design decisions.

This paper addresses a pressing need to understand the capabilities and performance characteristics of true CXL memory, as well as their impact on the performance of (co-running) applications and the design of OS policies to best use CXL memory. To this end, we take a system based on the CXL-ready 4th-generation Intel Xeon CPU [35] and three CXL memory devices from different manufacturers (§3). Then, for the first time, we not only compare the performance of true CXL memory with that of emulated CXL memory, but also conduct an in-depth analysis of the complex interplay between the CPU and CXL memory. Based on these comprehensive analyses, we make the following contributions.

CXL memory ≠ remote NUMA memory (§4). We reveal that true CXL memory exhibits notably different performance Characteristics from emulated CXL memory. **(C1)** Depending on CXL controller designs and/or memory technologies, true CXL memory devices give a wide range of memory access latency and bandwidth values. **(C2)** True CXL memory can give up to 26% lower latency and 3–66% higher bandwidth efficiency than emulated CXL memory, depending on memory access instruction types and CXL memory devices. This is because true CXL memory has neither caches nor CPU cores that modify caches, although it is exposed as a NUMA node. As such, the CPU implements an on-chip hardware structure to facilitate fast cache coherence checks for memory accesses to the true CXL memory. These are important differences that may change conclusions made by prior work on the performance characteristics of CXL memory and consequently the effectiveness of the proposals at the system level. **(C3)** The sub-NUMA clustering (SNC) mode provides LLC isolation among SNC nodes (§3) by directing the CPU cores within an SNC node to evict their L2 cache lines from its local memory exclusively to LLC slices within the same SNC node. However, when CPU cores access CXL memory, they end up breaking the LLC isolation, as L2 cache lines from CXL memory can be evicted to LLC slices in any SNC nodes. Consequently, accessing CXL memory can benefit from effectively 2–4× larger LLC capacity than accessing local DDR memory, notably compensating for the longer latency of accessing CXL memory for cache-friendly applications.

Naïvely used CXL memory considered harmful (§5). Using a system with a CXL memory device, we evaluate a set of applications with diverse memory access characteristics and different performance metrics (*e.g.*, response time and throughput). Subsequently, we present the following Findings. **(F1)** Simple applications (*e.g.*, key-value-store) demanding μs-scale latency are highly sensitive to memory access latency. Consequently, allocating pages to CXL memory increases the tail latency of these applications by 10–82% compared to local DDR memory. Besides, the state-of-the-art CXL-memory-aware page placement policy for a tiered memory

system [64] actually increases tail latency even further when compared to statically partitioning pages between DDR memory and CXL memory. This is due to the overhead of page migration. **(F2)** Complex applications (*e.g.*, social network microservices) exhibiting *m*s-scale latency experience a marginal increase in tail latency even when most of pages are allocated to CXL memory. This is because the longer latency of accessing CXL memory contributes marginally to the end-to-end latency of such applications. **(F3)** Even for memory-bandwidth-intensive applications, naïvely allocating 50% of pages to CXL memory based on the default OS policy may result in lower throughput, despite higher aggregate bandwidth delivered by using both DDR memory and CXL memory.

CXL-memory-aware dynamic page allocation policy (§6). To showcase the usefulness of our characterizations and findings described above, we propose `Caption`, a **C**XL-memory-**a**ware dynamic **p**age alloca**tion** policy for the OS to more efficiently use the bandwidth expansion capability of CXL memory. Specifically, `Caption` begins by determining the bandwidth of manufacturer-specific CXL memory devices. Subsequently, `Caption` periodically monitors various CPU counters, such as memory access latency experienced by (co-running) applications and assesses the bandwidth consumed by them at runtime. Lastly, based on the monitored CPU counter values, `Caption` estimates memory-subsystem performance over periods. When a given application demands an allocation of new pages, `Caption` considers the history of memory subsystem performance and the percentage of pages allocated to CXL memory in the past. Then, it adjusts the percentage of the pages allocated to CXL memory to improve the overall system throughput using a simple greedy algorithm. Our evaluation shows that `Caption` improves the throughput of a system co-running a set of memory-bandwidth-intensive SPEC CPU2017 benchmarks by 24%, compared with the default static page allocation policy set by the OS.

2 BACKGROUND

2.1 Compute eXpress Link (CXL)

PCIe is the industry standard for a high-speed serial interface between a CPU and I/O devices. Each lane of the current PCIe 5.0 can deliver 32 GT/s (*e.g.*, ~64 GB/s with 16 lanes). Built on the physical layer of PCIe, the CXL standard defines three separate protocols: `CXL.io`, `CXL.cache`, and `CXL.mem`. `CXL.io` uses protocol features of the standard PCIe, such as transaction-layer packet (TLP) and data-link-layer packet (DLLP), to initialize the interface between a CPU and a device [20]. `CXL.cache` and `CXL.mem` use the aforementioned protocol features for the device to access the CPU's memory and for the CPU to access the device's memory, respectively.

The `CXL.mem` protocol accounts only for memory accesses from the CPU to the device facilitated by the Home Agent (HA) and the CXL controller on the CPU and the device, respectively [70]. The HA handles the `CXL.mem` protocol and transparently exposes CXL memory to the CPU as memory in a remote NUMA node. That is, the CPU can access CXL memory with load and store instructions in the same way as it accesses memory in a remote NUMA node. This has an advantage over other memory expansion technologies, such as RDMA, which involves the device's DMA engine and thus has different memory access semantics. Lastly, when the CPU accesses

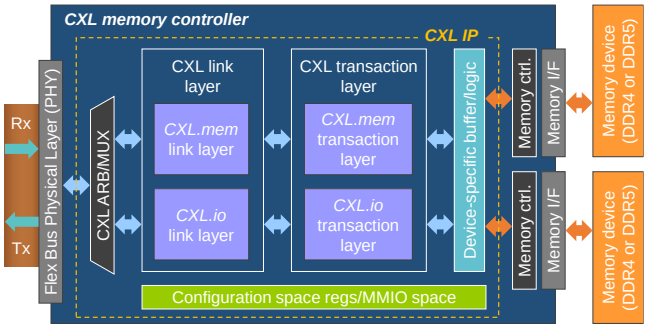


Figure 2: CXL.mem controller architecture.

CXL memory, it caches data from/to the CXL memory in every level of its cache hierarchy. This has been impossible with any other memory extension technologies except for persistent memory.

2.2 CXL-ready Systems and Memory Devices

CXL requires hardware support from both the CPU and devices. Both the latest 4th-generation Intel Xeon Scalable Processor (Sapphire Rapids) and the latest 4th-generation AMD EPYC Processor (Genoa) are among the first server-class commodity CPUs to support the CXL 1.1 standard [4, 35]. Figure 2 depicts a typical architecture of `CXL.mem` controllers. It primarily consists of (1) PCIe physical layer, (2) CXL link layer, (3) CXL transaction layer, and (4) memory controller blocks. (2), (3), and other CXL-related components are collectively referred to as CXL IP in this paper. As of today, in addition to some research prototypes, multiple CXL memory devices have been designed by major hardware manufacturers, such as Samsung [25], SK Hynix [77], Micron [65], and Montage [67]. To facilitate more flexible memory functionality and near-memory computing capability, Intel also enables the CXL protocol in its latest Agilex-I series FPGA [40], integrated with hard CXL IPs to support the `CXL.io`, `CXL.cache`, and `CXL.mem` [41]. Lastly, unlike a true NUMA node typically based on a large server-class CPU, a CXL memory device does not have any CPU cores, caches, or long interconnects between the CXL IP and the memory controller in the device.

3 EVALUATION SETUP

3.1 System and Device

Systems. We use a server to evaluate the latest commercial hardware supporting CXL memory (Table 1). The server consists of two Intel Sapphire Rapids (SPR) CPU sockets. One socket is populated with eight 4800 MT/s DDR5 DRAM DIMMs (128 GB) across eight memory channels. The other socket is populated with only one 4800 MT/s DDR5 DRAM DIMM to emulate the bandwidth and capacity of CXL memory. The Intel SPR CPU integrates four CPU chiplets, each with up to 15 cores and two DDR5 DRAM channels. A user can choose to use the 4 chiplets as a unified CPU, or each chiplet (or two chiplets) as a NUMA node in the SNC mode. Such flexibility is to give users strong isolation of shared resources, such as LLC, among applications. Lastly, we turn off the hyper-threading feature and set the CPU core clock frequency to 2.1 GHz for more predictable performance.

Table 1: System configurations.

Dual-socket server system	
Component	**Desription**
OS (kernel)	Ubuntu 22.04.2 LTS (Linux kernel v6.2)
CPU	2× Intel® Xeon 6430 CPUs @2.1 GHz [37], 32 cores and 60 MB LLC per CPU, Hyper-Threading disabled
Memory	Socket 0: 8× DDR5-4800 channels Socket 1: 1× DDR5-4800 channel (emulated CXL memory)

CXL memory devices			
Device	**CXL IP**	**Memory technology**	**Max. bandwidth**
CXL-A	Hard IP	DDR5-4800	38.4 GB/s per channel
CXL-B	Hard IP	2× DDR4-2400	19.2 GB/s per channel
CXL-C	Soft IP	DDR4-3200	25.6 GB/s per channel

CXL memory devices. We take three CXL memory devices ('CXL memory devices' in Table 1), each featuring different CXL IPs (ASIC-based hard IP and FPGA-based soft IP) and DRAM technologies (DDR5-4800, DDR4-2400, and DDR4-3200). Since the CXL protocol itself does not prescribe the underlying memory technology, it can seamlessly and transparently accommodate not only DRAM but also persistent memory, flash [72], and other emerging memory technologies. Consequently, various CXL memory devices may exhibit different latency and bandwidth characteristics.

3.2 Microbenchmark

To characterize the performance of CXL memory, we use two microbenchmarks. First, we use Intel Memory Latency Checker (MLC) [42], a tool used to measure memory latency and bandwidth for various usage scenarios. Second, we use a microbenchmark dubbed `memo` (**m**easuring **e**fficiency of **m**em**o**ry subsystems). It shares some features with Intel MLC, but we develop it to give more control over characterizing memory subsystem performance in diverse ways. For instance, it can measure the latency and bandwidth of a specific memory access instruction (*e.g.*, AVX-512 non-temporal load and store instructions).

3.3 Benchmark

Latency-sensitive applications. We run `Redis` [69], a popular high-performance in-memory key-value store, with `YCSB` [19]. We use a uniform distribution of keys, ensuring maximum stress on the memory subsystem, unless we explicitly specify the use of other distributions. We also run DeathStarBench (`DSB`) [28], an open-source benchmark suite designed to evaluate the performance of microservices. It uses `Docker` to launch components of a microservice, including machine learning (ML) inference logic, web backend, load balancer, caching, and storage. Specifically, we evaluate three `DSB` workloads: (1) `compose posts`, (2) `read user timelines`, and (3) `mixed workloads` (10% of `compose posts`, 30% of `read user timelines`, and 60% of `read home timelines`) as a social network framework. Lastly, we run `FIO` [7], an open-source tool used for benchmarking storage devices and file systems, to evaluate the latency impact of using CXL memory for OS page cache. The page cache is supported by the standard Linux storage subsystem, which holds recently accessed storage data (*e.g.*, files) in unused main memory space to reduce the number of accesses to slow storage devices.

Throughput applications. First, we run an inference application based on a deep learning recommendation model (`DLRM`) with the same setup as `MERCI` [58]. The embedding reduction step in DLRM inference is known to have a large memory footprint and is responsible for 50–70% of the inference latency [58]. Second, we take the SPECrate CPU2017 benchmark suite [13], which is commonly used to evaluate the throughput of systems in datacenters. Then we assess misses per kilo instructions (MPKI) of every benchmark and run the four benchmarks with the highest MPKI: (1) `fotonik3d`, (2) `mcf`, (3) `roms`, and (4) `cactuBSSN`. We run multiple instances of a single benchmark or two different benchmarks.

4 MEMORY LATENCY AND BANDWIDTH CHARACTERISTICS

In this section, we first evaluate the latency and bandwidth of accessing different memory devices: an emulated CXL memory device based on DDR5 memory in a remote NUMA node (DDR5-R), and three true CXL memory devices (CXL-A, CXL-B, and CXL-C). We conduct this evaluation to understand the performance characteristics of various CXL memory devices for different memory access instruction types. Next, we investigate interactions between the Intel SPR CPU's cache hierarchy and the CXL memory devices.

4.1 Latency

Figure 3 presents the measured latency values of accessing both emulated and true CXL memory devices. The first group of bars shows average (unloaded idle) memory access latency values measured by Intel MLC that performs pointer-chasing (*i.e.*, getting the memory address of a load from the value of the preceding load) in a memory region larger than the total LLC capacity of the CPU. This effectively measures the latency of serialized memory accesses. The remaining four groups of bars show the average memory access latency values measured by `memo` for four memory access instruction types: (1) temporal load (`ld`), (2) non-temporal load (`nt-ld`), (3) temporal store (`st`), and (4) non-temporal store (`nt-st`). For these groups, we first execute `clflush` to flush all cache lines from the cache hierarchy and then `mfence` to ensure the completion of flushing the cache lines. Then, we execute 16 memory access instructions back to back to 16 random memory addresses in a cacheable memory region. To measure the execution time of these 16 memory access instructions, we execute `rdtsc`, which reads the current value of the CPU's 64-bit time-stamp counter into a register immediately before and after executing the 16 memory

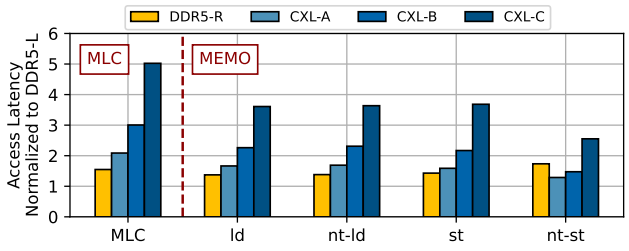


Figure 3: Random memory access latency of various memory devices (DDR5-R, CXL-A, CXL-B, and CXL-C), measured by Intel MLC and `memo`.

access instructions, followed by an appropriate fence instruction. This effectively measures the latency of random parallel memory accesses for each memory access instruction type for a given memory device. To obtain a representative latency value, we repeat the measurement 10,000 times and choose the median value to exclude outliers caused by TLB misses and OS activities.

Analyzing the latency values shown in Figure 3, we make the following **O**bservations.

(O1) The full-duplex CXL and UPI interfaces reduce memory access latency. `memo` gives emulated CXL memory 76% lower `ld` latency than Intel MLC. This difference arises because serialized memory accesses by Intel MLC cannot exploit the full-duplex capability of the UPI interface that connects NUMA nodes. In contrast, random parallel memory accesses by `memo` can send memory commands/addresses and receive data in parallel through the full-duplex UPI interface, effectively halving the average latency cost of going through the UPI interface. Since true CXL memory is also based on the full-duplex interface (*i.e.*, PCIe), it enjoys the same benefit as emulated CXL memory. Nonetheless, with `memo`, CXL-A gets a 3 percentage points more `ld` latency reduction than for DDR5-R. (O3) explains the reason for this additional latency reduction.

(O2) The latency of accessing true CXL memory devices is highly dependent on a given CXL controller design. Figure 3 shows that CXL-A exhibits only 35% longer `ld` latency than DDR5-R, while CXL-B and CXL-C present almost 2× and 3× longer `ld` latency, respectively. Even with the same DDR4 DRAM technology, CXL-C based on DDR4-3200 gives 67% longer `ld` latency than CXL-B based on DDR4-2400.

(O3) Emulated CXL memory can give longer memory access latency than true CXL memory. When issuing memory requests to emulated CXL memory, the local CPU must first check with the remote CPU, which is connected through the inter-chip UPI interface, for cache coherence [62, 66]. Moreover, the memory requests must travel through a long intra-chip interconnect within the remote CPU to reach its memory controllers [83]. These overheads increase with more CPU cores, *i.e.*, more caches and a longer interconnect path. For example, the `ld` latency values of DDR5-R with 26- and 40-core Intel SPR CPUs are 29% lower and 19% higher, respectively, than those of DDR5-L with the 32-core Intel SPR CPU used for our primary evaluations. In contrast, true CXL memory has neither caches nor CPU cores that modify caches, although it is exposed as a remote NUMA node. As such, the CPU implements an on-chip hardware structure to facilitate fast cache coherence checks for memory accesses to the true CXL memory. Moreover, true CXL memory features a short intra-chip interconnect within the CXL controller to reach its memory controllers.

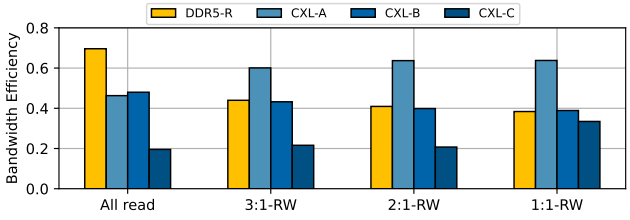
Specifically for DDR5-R and CXL-A, `memo` provides 76% and 79% lower `ld` latency values, respectively, than Intel MLC. Although both DDR5-R and CXL-A benefit from (O1), CXL-A gives a more `ld` latency reduction than DDR5-R. This arises from the following differences between `memo` and Intel MLC. As Intel MLC accesses memory serially, the local CPU performs the aforementioned cache coherence checks one by one. By contrast, `memo` accesses memory in parallel. In such a case, memory accesses to emulated CXL memory incur a burst of cache coherence checks that need to go through the inter-chip UPI interface, leading to cache coherence traffic congestion. This, in turn, increases the time required for cache coherence checks. However, memory accesses to true CXL memory suffer notably less from this overhead because the CPU checks cache coherence through its local on-chip structure described earlier. This contributes to a further reduction in the `ld` latency for true CXL memory. Also note that DDR5-R, based on a 40-core Intel SPR CPU, presents 4% longer `ld` latency than CXL-A due to a higher overhead of cache coherence checks.

The overhead of cache coherence checks becomes even more prominent for `st` because of two reasons. First, the `st` latency is much higher than the `ld` latency in general. For example, the latency of `st` to DDR5-R is 2.3× longer than that of `ld` from DDR5-L. This is because of the cache write-allocate policy in Intel CPUs. When `st` experiences an LLC miss, the CPU first reads 64-byte data from memory to a cache line (*i.e.*, implicitly executing `ld`), and then writes modified data to the cache line [43]. This overhead is increased for both emulated CXL memory and true CXL memory, as the overhead of traversing the UPI and CXL interfaces is doubled. Yet, the latency of `st` to emulated CXL memory increases more than that of `st` to emulated CXL memory, when compared to `ld` or `nt-ld`. This is because the emulated CXL memory incurs a higher overhead for cache coherence checks than the true CXL memory, as discussed earlier.

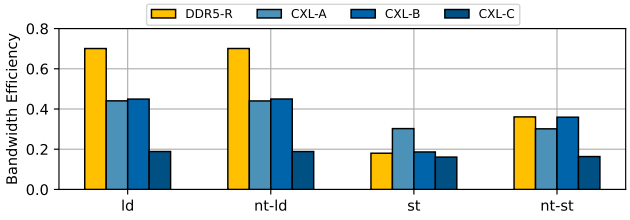
Lastly, although both `nt-ld` and `nt-st` bypass caches and directly access memory, the local CPU accessing emulated CXL memory still needs to check with the remote CPU for cache coherence [39]. This explains why the `nt-ld` latency values of all the memory devices are similar to those of `ld` that needs to be served by memory in Figure 3. Unlike `st`, however, `nt-st` does not read 64-byte data from the memory since it does not allocate a cache line by its semantics, which eliminates the memory access and cache coherence overheads associated with implicit `ld`. Therefore, the absolute values of `nt-st` latency across all the memory devices are smaller than those of `st` latency. Furthermore, `nt-st` can also offer shorter latency than `ld` and `nt-ld` because the CPU issuing `nt-st` sends the address and data simultaneously. In contrast, the CPU issuing `ld` or `nt-ld` sends the address first and then receive the data later, which makes signals go through the UPI and CXL interfaces twice. With shorter latency for a cache coherency check, `nt-st` to true CXL memory can be shorter than `nt-st` to emulated CXL memory. For instance, CXL-A exhibits a 25% lower latency than DDR5-R for `nt-st`. Note that `nt-st` behaves differently depending on whether the allocated memory region is cacheable or not [39], and we conduct our experiment with a cacheable memory region.

4.2 Bandwidth

The sequential memory access bandwidth represents the maximum throughput of the memory subsystem when all the CPU cores sends memory requests in this paper. Nonetheless, it notably varies across (1) CXL controller designs, (2) memory access instruction types, and (3) DRAM technologies (*i.e.*, DDR5-4800, DDR4-3200, and DDR4-2400 in this paper). As such, for fair and insightful comparison, we use bandwidth efficiency as a metric, normalizing the measured bandwidth to the theoretical maximum bandwidth. Figure 4



(a) MLC with various read and write ratios



(b) `memo` with different memory instruction types

Figure 4: Efficiency of maximum sequential memory access bandwidth across different memory types.

presents the bandwidth efficiency values of DDR5-R, CXL-A, CXL-B, and CXL-C for various read/write ratios and different memory instruction types, respectively. Analyzing these values, we make the following **O**bservations.

(O4) The bandwidth is strongly dependent on the efficiency of CXL controllers. The maximum sequential bandwidth values that can be provided by the DDR5-4800, DDR4-3200, and DDR4-2400 DRAM technologies are 38.4 GB/s, 25.6 GB/s, and 19.2 GB/s per channel, respectively. Nonetheless, Figure 4a shows that DDR5-R, CXL-A, CXL-B, and CXL-C provides only 70%, 46%, 47%, and 20% of the theoretical maximum bandwidth, respectively, for 'All Read'. DDR5-R and CXL-A are based on the same DDR5-4800 DRAM technology, yet the bandwidth efficiency of DDR5-R is 23 percentage points higher than that of CXL-A. We speculate that the lower efficiency of the CXL-A's memory controller for memory read accesses contributes to this bandwidth efficiency gap, as both DDR5-R and CXL-A exhibit similar `ld` latency values.

As the write ratio increases, however, CXL-A starts to provide higher bandwidth efficiencies. For example, Figure 4a shows that the bandwidth efficiency of CXL-A for '2:1-RW' is 23 percentage points higher than that of DDR5-R. We speculate that the CXL-A's memory controller is designed to handle interleaved memory read and write accesses more efficiently than the DDR5-R's and CXL-B's memory controllers. This is supported by (1) the fact that `st` involves both memory read and write accesses due to implicit `ld` when it incurs a cache miss (*cf.* (O3)), and (2) the bandwidth efficiency of CXL-A for all the other memory access instruction types is lower than that of DDR5-R and CXL-B. This also implies that the higher bandwidth efficiency of CXL-A for `st` is not solely attributed to a unique property of true CXL memory.

Figure 4b shows that the bandwidth efficiency of CXL-B is higher than that of CXL-A, except for `st`, although the latency values of CXL-B is higher than those of CXL-A. Specifically, the bandwidth efficiency values of CXL-B for `ld`, `nt-ld`, and `nt-st` are 1, 1, and 6 percentage points higher than that of CXL-A, respectively. We speculate that the recently developed third-party DDR5 memory controllers may not be as efficient as the mature and highly optimized DDR4 memory controller used in CXL-B for read- or write-only memory accesses. Note that CXL-C is based on DDR4-3200 DRAM technology, but it generally exhibits poor bandwidth efficiency due to the FPGA-based implementation of the CXL controller. The bandwidth efficiency values of CXL-C for `ld`, `nt-ld`, `st`, and `nt-st` are 26, 26, 3, and 20 percentage points lower, respectively, than those of CXL-B, which is based on the same DDR4 DRAM technology but provides 25% lower theoretical maximum bandwidth per channel.

(O5) True CXL memory can offer competitive bandwidth efficiency for the store compared to emulated CXL memory. Figure 4b shows that `st` yields lower bandwidth efficiency values than `ld` across all the memory devices due to the overheads of implicit `ld` and cache coherence checks (*cf.* (O3)). Specifically, `st` to DDR5-R, CXL-A, CXL-B, and CXL-C offers 74%, 31%, 59%, and 15% lower bandwidth efficiency values, respectively, than `ld` from DDR5-R, CXL-A, CXL-B, and CXL-C. This suggests that emulated CXL memory experiences a notably more bandwidth efficiency degradation than true CXL memory partly because it suffers more from the overhead of cache coherence checks. As a result, the bandwidth efficiency values of CXL-A and CXL-B for `st` are 12 and 1 percentage points higher, respectively, than DDR5-R. For `nt-st`, the bandwidth efficiency gap between emulated CXL memory and true CXL memory is noticeably reduced compared to `nt-ld`. Specifically, the bandwidth efficiency gap between DDR5-R and CXL-A for `nt-ld` is 26 percentage points, whereas it is reduced to 6 percentage points for `nt-st`. CXL-B provides almost the same bandwidth efficiency as DDR5-R for `nt-st`.

4.3 Interaction with Cache Hierarchy

Starting with the Intel Skylake CPU, Intel has adopted non-inclusive cache architecture [34]. Suppose that a CPU core with non-inclusive cache architecture incurs an LLC miss that needs to be served by memory. Then it loads data from the memory into a cache line in the the CPU core's (private) L2 cache rather than the (shared) LLC,

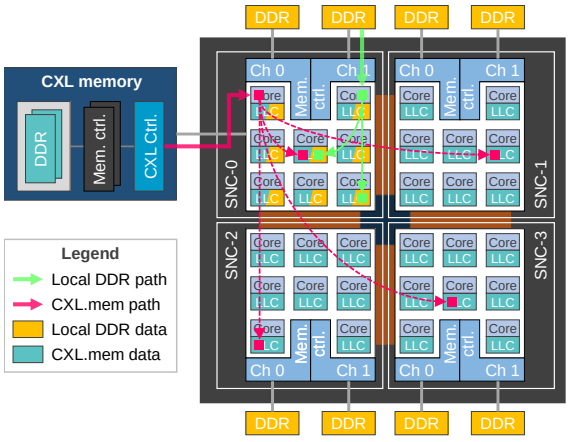
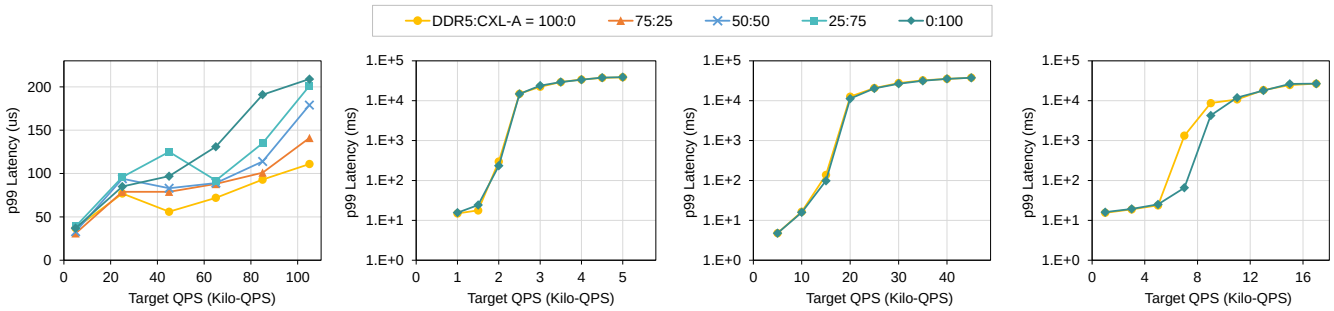


Figure 5: Difference in L2 cache line eviction paths between local DDR memory and CXL memory in SNC mode.



(a) [Redis] YCSB-A (RD:UPD = 5:5) (b) [DSB] compose posts (c) [DSB] read user timelines (d) [DSB] mixed workloads

Figure 6: 99th-percentile (p99) latency values of `Redis` with various percentages of pages allocated to CXL memory and `DSB` with 100% of only 'caching and storage' pages allocated to either CXL memory or DDR memory.

in contrast to a CPU core with inclusive cache architecture. When evicting the cache line in the L2 cache, the CPU core will place it into the LLC. That is, the LLC serves as a victim cache. The SNC mode (§3.1), however, restricts where the CPU core places evicted L2 cache lines within the LLC to provide LLC isolation among SNC nodes. The LLC comprises as many slices as the number of CPU cores, and L2 cache lines in an SNC node are evicted only to LLC slices within the same SNC node when data in the cache lines are from the local DDR memory of that SNC node (light-green lines in Figure 5). In contrast, we notice that L2 cache lines can be evicted to any LLC slices within any SNC nodes when the data are from remote memory, including both emulated CXL memory and true CXL memory (red-dashed lines in Figure 5). As such, CPU cores accessing CXL memory break LLC isolation among SNC nodes in SNC mode. This makes such CPU cores benefit from 2–4× larger LLC capacity than the ones accessing local DDR memory, notably compensating for the slower access latency of CXL memory.

To verify this, we run a single instance of Intel MLC on an idle CPU and measure the average latency of randomly accessing 32 MB buffers allocated to DDR5-L and CXL-A, respectively, in the SNC mode. The total LLC capacity of the CPU with four SNC nodes in our system is ~60 MB. A 32 MB buffer is larger than the total LLC capacity of a single SNC node but smaller than that of all four SNC nodes. This shows that accessing the buffer allocated to CXL-A gives an average memory access latency of 41 ns, whereas accessing the buffer allocated to DDR5-L offers an average memory access latency of 76.8 ns. The shorter latency of accessing the buffer allocated to CXL-A evidently shows that the CPU cores accessing CXL memory can benefit from larger effective LLC capacity than the CPU cores accessing local DDR memory in the SNC mode.

(O6) CXL memory interacts with the CPU's cache hierarchy differently compared to local DDR memory. As discussed above, the CPU cores accessing CXL memory are exposed to a larger effective LLC capacity in the SNC mode. This often significantly impacts LLC hit/miss and interference characteristics that the CPU cores experience, and thus affecting the performance of applications (§5.3). Therefore, we must consider this attribute of CXL memory when analyzing the performance of applications using CXL memory, especially in the SNC mode.

5 IMPACT OF USING CXL MEMORY ON APPLICATION PERFORMANCE

To study the impact of using CXL memory on the performance of applications (§3.3), we take CXL-A, which provides the most balanced latency and bandwidth characteristics among the three CXL memory devices. We use `numactl` in Linux to allocate memory pages of a given program, either fully or partially, to CXL memory, exploiting the fact that the OS recognizes CXL memory as memory in a remote NUMA node by the OS. Specifically, `numactl` allows users to: (1) bind a program to a specific memory node (`membind` mode); (2) allocate memory pages to the local NUMA node first, and then other remote NUMA nodes only when the local NUMA node runs out of memory space (`preferred` mode); or (3) allocate memory pages evenly across a set of nodes (`interleaved` mode). A recent Linux kernel patch [85] enhances the `interleaved` mode to facilitate fine-grained control over the allocation of a specific percentage of pages to a chosen NUMA node. For example, we can change the percentage of pages allocated to CXL memory from the default 50% to 25%. That is, 75% of pages are allocated to local DDR5 memory.

In this section, we will vary the percentage of pages allocated to CXL memory and analyze its impact on performance using application-specific performance metrics, setting the stage for our CXL memory-aware dynamic page allocation policy (§6). Note that we enable the SNC mode to use only two local DDR5 memory channels along with one CXL memory channel. This is because our system can accommodate only one CXL memory device, and it needs to make a meaningful contribution to the total bandwidth of the system. In such a setup, the local DDR5 memory with two channels provides ~2× higher bandwidth for `st` and ~3.4× higher bandwidth for `ld` than CXL memory. As future platforms accommodate more CXL memory devices, we may connect up to four CXL memory devices to a single CPU socket with eight DDR5 memory channels, providing the same DDR5 to CXL channel ratio as our setup.

5.1 Latency

Redis. Figure 6a shows the 99th-percentile (p99) latency values of `Redis` with `YCSB` workload `A` (50% read and 50% update) while

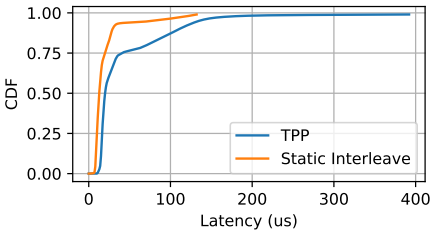


Figure 7: Impact of TPP on latency of `Redis` compared with statically allocating 25% of (random) pages to CXL memory. We show the distributions up to the p99 latency.

varying the percentage of pages allocated to CXL memory or local DDR5 memory (referred to as DDR memory hereafter). First, allocating 100% of pages to CXL memory (CXL 100%) significantly increases the p99 latency compared to allocating 100% of pages to DDR memory (DDR 100%), especially at high target QPS values. For example, CXL 100% results in 10%, 73%, and 105% higher p99 latency than DDR 100% at 25 K, 45 K, and 85 K target QPS, respectively. Second, as more pages are allocated to CXL memory, the p99 latency increases proportionally. For instance, at 85 K target QPS, allocating 25%, 50%, and 75% of pages to CXL memory results in p99 latency increases of 9%, 23%, and 45%, respectively, compared to DDR 100%. Finally, as expected, allocating 100% of pages to DDR memory results in the lowest and most stable p99 latency. Explaining the substantial difference in the p99 latency values for various percentages of pages allocated to CXL memory and different target QPS values, we note that `Redis` typically operates with response time at a μs scale, making it highly sensitive to memory access latency (§3.3). Therefore, allocating more pages to CXL memory and/or increasing the target QPS makes `Redis` more frequently access CXL memory with almost 2× longer latency than DDR memory, resulting in higher p99 latency.

Redis+TPP. We conduct an experiment to assess whether the most recent transparent page placement (TPP) [64] can minimize the impact of using CXL memory on the p99 latency. The most recent publicly available release [63] only offers an enhanced page migration policy, and it does not automatically place pages in CXL memory. Thus, we begin by allocating 100% of pages requested by `Redis` to CXL memory and let TPP automatically migrate pages to DDR memory until the percentage of the pages allocated to CXL memory becomes 25%, based on the DDR to CXL bandwidth ratio in our setup. Then we measure the latency values of `Redis`. Figure 7 compares two distributions of the measured latency values. The first one is from using TPP, while the second one is from statically allocating 25% of pages to CXL memory.

TPP migrates a large number of pages to DDR memory in the beginning phase, requiring the CPU to (1) copy pages from one memory device to another and (2) update the OS page table entries associated with the migrated pages [89]. Since (1) and (2) incur high overheads, we measure the p99 latency only after 75% of pages are migrated to DDR memory. As shown in Figure 7, TPP generally gives higher latency, resulting in 174% higher p99 latency than statically allocating 25% of pages to CXL memory. This is because TPP constantly migrates a small percentage of pages between DDR memory and CXL memory over time, based on its metric assessing hotness/coldness of the pages. Although TPP has a feature that reduces ping-pong behavior (*i.e.*, pages are constantly being promoted and demoted between DDR memory and CXL memory), migrating pages incurs the overheads from (1) and (2) above. (1) blocks the memory controllers from serving urgent memory read requests from latency-sensitive applications [57], and (2) also requires a considerable number of CPU cycles and memory accesses.

Table 2: Components of `DSB` social network benchmark.

Name	Working set	Intensiveness	Allocated mem. type
Frontend	83 MB	Compute	DDR memory
Logic	208 MB	Compute	DDR memory
Caching & Storage	628 MB	Memory	CXL memory

DSB. Figure 6b–6d present the p99 latency values of (b) `compose posts`, (c) `read user timelines`, and (d) `mixed workloads`. Table 2 summarizes the components of the benchmarks, their working set sizes and characteristics, and allocated memory devices. In our experiment, we allocate 100% of the pages pertinent to the caching and storage components with large working sets to either DDR memory (DDR 100%) or CXL memory (CXL 100%). Meanwhile, we always allocate 100% of the pages associated with the remaining components, such as `nginx` front-end and analytic docker images (*e.g.*, logic in Table 2), to DDR memory, since these components are more sensitive to memory access latency than the caching and storage components. For example, `nginx` spends 60% of CPU cycles on the CPU front-end, which is dominated by fetching instructions from memory [28]. Therefore, pages of such components should be allocated to DDR memory.

This experiment shows that all three benchmarks, `compose posts`, `read user timelines`, and `mixed workloads` are not sensitive to long latency of accessing CXL memory as they exhibit little difference in p99 latency values between CXL 100% and DDR 100%. This is because of two reasons. First, most of the p99 latency in these benchmarks is contributed by the front-end and/or logic components (*i.e.*, DDR 100%). This makes the latency of accessing CXL memory amortized by the these components, and thus the p99 latency is much less dependent on the latency of accessing databases (*i.e.*, CXL 100%). Second, the p99 latency of `DSB` is at a *ms* scale and two orders of magnitude longer than that of `Redis`. Therefore, it is not as sensitive to memory access latency as that of `Redis`.

Note that CXL 100% provides lower p99 latency values than DDR 100% for `mixed workloads` when the QPS range is between 5 K and 11 K. This is because `mixed workloads` is far more memory-bandwidth-intensive than `compose posts` and `read user timelines`. Specifically, when we measure the average bandwidth consumption by these three benchmarks in the QPS range that saturates the throughput of the benchmarks, we observe that `mixed workloads` consumes 32 GB/s while `compose posts` and `read user timelines` consume only 7 GB/s and 10 GB/s, respectively. When a given application consumes such high bandwidth in our setup, we observe that the application's throughput, which is inversely proportional to its latency, becomes sensitive to the bandwidth available for the application (§5.2). Lastly, as the QPS approaches to 11 K, the compute capability of the CPU cores becomes the dominant bottleneck, leading to a decrease in the p99 latency gap between DDR 100% and CXL 100%.

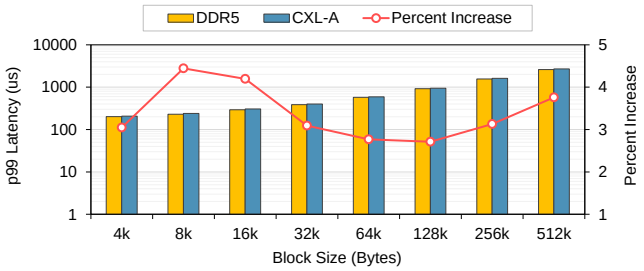


Figure 8: p99 latency values of `FIO` for various block sizes, and percentage values of increase in p99 latency by allocating page cache to CXL.

FIO. Figure 8 presents the p99 latency values of `FIO` with 4 GB page cache allocated to either DDR memory or CXL memory for various I/O block sizes. We use a `Zipfian` distribution for `FIO` to evaluate the impact of using page cache on file-system performance. It shows that allocating the page cache to CXL memory gives only ~3% longer p99 latency than DDR5 memory for 4 KB block size. This is because the p99 latency for a 4 KB block size is primarily dominated by the Linux kernel operations related to page cache management, such as context switching, page cache lookup, sending an I/O request through the file system, block layer, and device driver. However, with a 8 KB block size, the cost of Linux kernel operations is amortized, as multiple 4 KB pages are brought from a storage device by a single system call. Consequently, longer access latency of CXL memory affects the p99 latency of `FIO` more notably, resulting in a ~4.5% increase in the p99 latency. Meanwhile, as the block size increases beyond 8 KB, the page cache hit rate decreases from 76% for 8 KB to 65% for 128 KB. As lower page cache hit rates necessitate more page transfers from the storage device, the storage access latency begins to dominate the p99 latency. In such a case, the difference in memory access latency between DDR memory and CXL memory exhibits a lower impact on p99 latency, since Data Direct I/O (DDIO) [38] directly injects pages read from the storage device into the LLC [3, 26, 27, 93]. Lastly, we observe another trend shift beyond 128 KB block size, which is mainly due to the limited read and write bandwidth of CXL memory. As more page cache entries are evicted from the LLC to memory as well as from memory to the storage device, the limited bandwidth of CXL memory increases the effective latency of I/O requests.

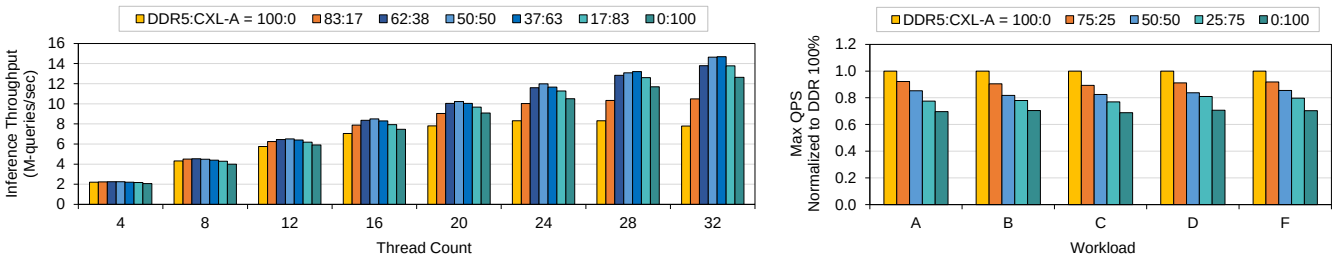
Key findings. Based on our analyses above, we present the following three key Findings. **(F1)** Allocating any percentage of pages to CXL memory proportionally increases the p99 latency of simple memory-intensive applications demanding μs-scale latency, since such applications are highly sensitive to memory access latency. **(F2)** Even an intelligent page migration policy may further increase the p99 latency of such latency-sensitive applications because of the overheads of migrating pages. **(F3)** Judiciously allocating certain pages to CXL memory does not increase the p99 latency of complex applications exhibiting *m*s-scale latency. This is because the long latency of accessing CXL memory marginally contributes to the end-to-end latency of such applications and it is amortized by intermediate operations between accesses to CXL memory.

5.2 Throughput

DLRM. Figure 9a shows the throughput of `DLRM` embedding reduction for various percentages of pages allocated to CXL memory. As the throughput of `DLRM` embedding reduction is bounded by memory bandwidth [50, 58, 94], it begins to saturate over 20 threads when 100% of pages are allocated to DDR memory. In such a case, we observe that allocating a certain percentage of pages to CXL memory can improve the throughput further, as it supplements to the bandwidth of DDR memory, increasing the total bandwidth available for `DLRM`. For instance, when running 32 threads, we observe that allocating 63% of pages to CXL memory can maximize the throughput of `DLRM` embedding reduction, providing 88% higher throughput than DDR 100%. Note that a lower percentage of pages will be allocated to CXL memory for achieving the maximum throughput if the maximum bandwidth capability of a given CXL memory device is lower (*e.g.*, CXL-C). This clearly demonstrates the benefit of CXL memory as a memory bandwidth expander.

Redis. Although `Redis` is a latency-sensitive application, its throughput is also an important performance metric. Figure 9b shows the maximum sustainable QPS for various percentages of pages allocated to CXL memory. For example, for `YCSB-A`, allocating 25%, 50%, 75%, and 100% of pages to CXL memory provides 8%, 15%, 22%, and 30% lower throughput than allocating 100% of pages to DDR memory. As `Redis` does not fully utilize the memory bandwidth, its throughput is bounded by memory access latency. Thus, similar to its p99 latency trends (Figure 6a), allocating more pages to CXL memory reduces the throughput of `Redis`.

Key findings. Based on our analyses above, we present the following key Finding. **(F4)** For memory-bandwidth-intensive applications, naïvely allocating 50% of pages to CXL memory based on the OS default policy may result in lower throughput than allocating 100% of pages to DDR memory, even with higher total bandwidth



(a) [DLRM] embedding reduction

(b) [Redis] YCSB-A

Figure 9: Impact of using CXL memory on throughput of `Redis` and `DLRM` for various ratios of page allocation to CXL memory.

Table 3: Throughput of DLRM using only 1 SNC node versus all 4 SNC nodes, normalized to the throughput of DLRM running on 1 SNC node allocating all the pages to local DDR memory.

1 SNC node		4 SNC nodes	
DDR 100%	CXL 100%	DDR 100%	CXL 100%
1	0.947	1	0.504

from using both DDR memory and CXL memory together. This motivates us to develop a dynamic page allocation policy that can automatically configure the percentage of pages allocated to CXL memory at runtime based on the bandwidth capability of a given CXL memory device and bandwidth consumed by co-running applications (§6).

5.3 Interaction with Cache Hierarchy

Previously, we discussed that accessing CXL memory breaks the LLC isolation among SNC nodes (§4.3). To analyze the impact of such an attribute of accessing CXL memory on application performance, we evaluate two cases. In the first case ('1 SNC node' in Table 3), only one SNC node (*i.e.*, SNC-0 in Figure 5) runs 8 DLRM threads while the other three SNC nodes idle. In the second case ('4 SNC nodes' in Table 3), each SNC node runs 8 DLRM threads. Only SNC-0 allocates 100% of its pages to either its DDR memory or CXL memory, while the other three SNC nodes (*i.e.*, SNC-1, SNC-2, and SNC-3) allocate 100% of their pages only to their respective local DDR memory. The second case is introduced to induce interference at the LLC among all the SNC nodes when SNC-0 with CXL 100%.

Table 3 shows that SNC-0 with CXL 100% in '1 SNC node' offers 88% higher throughput than SNC-0 with CXL 100% in '4 SNC nodes.' This is because of the other three SNC nodes in '4 SNC nodes' reduces the effective LLC capacity of SNC-0 with CXL 100% (*i.e.*, LLC slices from all the SNC nodes). Specifically, while the other three SNC nodes evict LLC lines within their respective LLC slices, they also inevitably evict many LLC lines from SNC-0 with CXL 100%. Although not shown in Table 3, SNC-0 with DDR 100% in '1 SNC node' provides 2% higher throughput than each of the other three SNC nodes in '4 SNC nodes' when SNC-0 in '4 SNC nodes' is with CXL 100%. This is because SNC-0 with CXL 100% in '4 SNC nodes' pollutes the LLC slices of the other three SNC nodes with cache lines evicted from the L2 caches of SNC-0, breaking the LLC isolation among the SNC nodes. Lastly, in our previous evaluation of DLRM throughput (§5.2), when SNC-0 needs to run more than 8 threads of DLRM in the SNC mode, it makes the remaining threads run on the CPU cores in the other three SNC nodes. Nonetheless, the CPU cores in the other three SNC nodes continue to access the DDR memory of SNC-0, and cache lines in the L2 caches of these CPU cores are still evicted to the LLC slices of SNC-0 since the cache lines were from the DDR memory of SNC-0.

6 CXL-MEMORY-AWARE DYNAMIC PAGE ALLOCATION POLICY

We have demonstrated a potential of CXL memory as a bandwidth expander, which can improve the performance of bandwidth-intensive applications (§5.2). If the throughput of a given application is limited by the bandwidth, allocating a higher percentage of pages

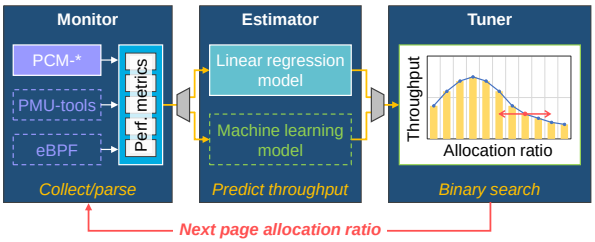


Figure 10: Overview of Caption. The components in dotted boxes can be used for better performance.

to CXL memory may alleviate bandwidth pressure on DDR memory, and hence reduce average memory access latency. Intuitively, such a percentage should be tuned for different CXL memory devices given their distinct bandwidth capabilities (§5.2). By contrast, if a given application is not memory-bandwidth-bounded, allocating a lower percentage of pages to CXL memory may lead to lower average memory access latency and thus higher throughput. That stated, to better utilize auxiliary memory bandwidth provided by CXL memory, we present Caption, a dynamic page allocation policy. Caption automatically tunes the percentage of new pages to be allocated by the OS to CXL memory based on three factors: (1) bandwidth capability of CXL memory, (2) memory intensiveness of co-running applications, and (3) average memory access latency. Note that Caption, focusing on the page allocation ratio between DDR memory and CXL memory, is orthogonal and complementary to TPP.

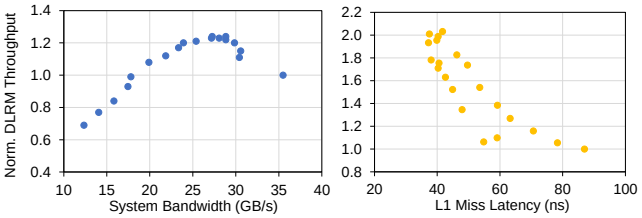
6.1 Policy Design

Caption consists of three runtime **M**odules (Figure 10). **(M1)** periodically monitors some CPU counters related to memory subsystem performance, and then **(M2)** estimates memory-subsystem performance based on values of the counters. When a given application requests an allocation of new pages, (M3) tunes the percentage of the new pages allocated to CXL memory, aiming to improve the throughput of the application. Subsequently, mempolicy [85] sets the page allocation ratio between DDR memory and CXL memory based on the percentage guided by (M3), and instructs the OS to allocate the new pages based on the ratio.

(M1) Monitoring CPU counters related to memory subsystem performance. We use Intel PCM [36] to periodically sample various CPU counters related to memory subsystem performance, as listed in Table 4. These CPU counters allow (M2) to estimate overall memory-subsystem performance. In Figure 11, we run DLRM, of which the throughput is bounded by memory bandwidth. Then we observe correlations between DLRM throughput and values of those counters, as we vary the percentage of pages allocated to CXL memory.

Table 4: CPU counters pertinent to memory-subsystem perf.

Metric	Tool	Description
L1 miss latency	pcm-latency	Average L1 miss latency (ns)
DDR read latency	pcm-latency	DDR read latency (ns)
IPC	pcm	Instructions per cycle



(a) System bandwidth **(b) L1 miss latency**

Figure 11: Correlations between throughput and various counter values, as we increase the percentage of pages allocated to CXL memory for DLRM. The system bandwidth is the total consumed memory bandwidth, and The throughput is normalized to DDR 100%.

Figure 11a shows that DLRM throughput is proportional to the consumed memory bandwidth. Yet, as the consumed memory bandwidth exceeds a certain amount, the memory access latency rapidly increases due to contention and resulting queuing delay at the memory controller [80], which, in turn, decreases the DLRM throughput.

Meanwhile, Figure 11b shows that DLRM throughput is inversely proportional to L1 miss latency. The L1 miss latency is an important memory-subsystem performance metric that simultaneously captures both the cache friendliness and bandwidth intensiveness (*i.e.*, queuing delay at the memory controller) of given (co-running) applications at the same time. At first, allocating more pages to CXL memory reduces pressure on the DDR memory controller, thereby decreasing the latency of accessing DDR memory and handling L1 misses. However, at some point, the long latency of accessing CXL memory begins to dominate that of handling L1 misses, and the application throughput begins to decrease. Finally, IPC is another important metric that implicitly measures the efficiency of the memory subsystem for the applications.

(M2) Estimating system throughput. To build a model that estimates the system performance, we collect CPU counter values at various DDR:CXL ratios while running DLRM with 24 threads. We then build a linear-regression model that correlates these counter values with DLRM throughput. Taking these counter values from (M1), Caption periodically estimates (or infers) memory-subsystem performance at runtime. In our current implementation of Caption, (M1) samples the counters every 1 second. To reduce the noise among the values, we collect a moving average of the past 5 samples for each counter. The averaged value is then fed into (M2) for performance estimation. Although we may use a machine-learning (ML) model, we use the following simple linear model for the current implementation of Caption:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \quad (1)$$

where Y represents the estimated memory-subsystem performance, X_n represents a counter value listed in Table 4, and β_n represents the X_n's weight obtained through multiple linear regression steps. This linear model is simple enough to be used by the OS at a low performance cost, yet effective enough to estimate the memory-subsystem performance. In the current implementation of Caption, we find that using PCM toolkit is sufficient. Nonetheless, we may use PMU tools and eBPF [24] to access more diverse counters, facilitating a more precise estimation of memory-subsystem performance.

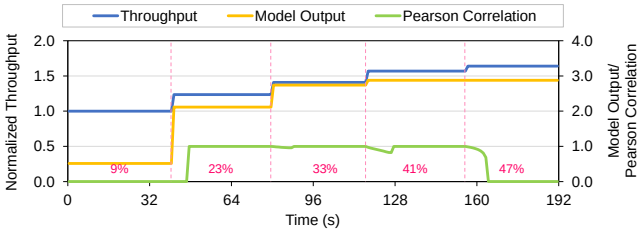
Algorithm 1: Caption tuning algorithm. state, step and ratio represent memory subsystem performance, unit of tuning page allocation ratio, and ratio of page allocation between DDR and CXL memory.

```
1  while true do
2      curr_state ← estimator()
3      if curr_state < prev_state then
4          curr_step ← prev_step × (−0.5)  // reverse
5      curr_ratio ← prev_ratio + curr_step
6      check_ratio_bound()
7      set_ratio(curr_ratio)
8      if new allocations then
9          prev_state ← curr_state
10         prev_step ← curr_step
11         prev_ratio ← curr_ratio
12     sleep(tune_interval)
```

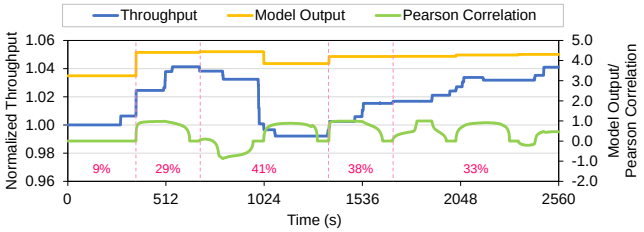
(M3) Tuning the percentage of pages allocated to CXL memory. When a given application demands allocation of memory pages, Caption (Algorithm 1) first compares the estimated memory-subsystem performance value from the past period (line: 9–11) with the current period (line: 3). If the memory-subsystem performance in the current period has increased compared to the previous period, Caption assumes that its previous decision, *i.e.*, increasing (or decreasing) the percentage of pages allocated to CXL memory, was correct. Then it continues to incrementally increase (or decrease) the percentage by a fixed amount (line: 5). Otherwise, it will begin to reverse the step by half (line: 4), which decreases (or increases) the percentage and evaluate the decision in the future period to determine a favorable percentage of pages allocated to CXL memory. Note that the absolute value of the step variable has the minimum limit (*e.g.*, 9% in our evaluation) to prevent it from being close to zero. Lastly, inspired by conventional control theory, Caption implements mechanisms to efficiently handle very small or sudden large changes in memory subsystem performance, even though they are not described in Algorithm 1.

6.2 Evaluation

We have developed Caption after analyzing the various performance characteristics and memory subsystem statistics of DLRM. However, we expect that Caption should work well for other applications because the monitored L1 miss latency, DDR read latency, and IPC counters are fundamental memory subsystem performance metrics that are strongly correlated with the throughput of memory-bandwidth-intensive applications; we believe CXL memory access latency and bandwidth statistics are also useful for estimating memory subsystem performance, but we currently cannot access the corresponding counters. To demonstrate this, we evaluate the efficacy of Caption by co-running (1) SPEC-Mix, various mixes of memory-intensive SPECrate CPU2017 benchmarks, and (2) Redis



(a) DLRM



(b) SPEC-Mix

Figure 12: Estimated memory-subsystem performance, measured application throughput, and Pearson coefficient values over time. The numbers represent the percentage values of pages allocated to CXL memory.

and DLRM without measuring their performance characteristics and memory-subsystem statistics in advance.

Figure 12 shows normalized measured throughput (Throughput), normalized estimated memory-subsystem performance (Eq. (1), Model Output), and Pearson correlation coefficient values. For DLRM we simply sweep the percentage of pages allocated to CXL memory over time. For SPEC-Mix, we let Caption automatically tune the percentage of pages allocated to CXL memory whenever a benchmark completes its execution. The Pearson correlation method allows us to quantify synchrony between time-series data [9]. The coefficient value can range from -1 and 1, indicating that both sets of data trend the same direction when it is positive. We calculate the Pearson correlation coefficient values to assess the effectiveness of the estimation model, as Algorithm 1 depends on precisely determining only the direction of performance changes after tuning the percentage of pages allocated to CXL memory. Figure 12 demonstrates that the Pearson correlation coefficient values mostly remains positive for both DLRM and SPEC-Mix. This indicates that the estimation model is adequate for Algorithm 1 to effectively tune both DLRM and SPEC-Mix. It is important to note that the estimation model is based on the weight values derived by fitting counter values from DLRM exclusively in this paper. However, it has the potential for further improvement by fitting counter values from a more diverse range of applications.

Figure 13 evaluates the efficacy of Caption for 16 instances of individual SPEC benchmarks, two different SPEC-Mix, and a mix of Redis and DLRM. For all the evaluation cases, Caption outperforms both 100% and 50% allocations to DDR memory while allocating substantial percentages of pages to CXL memory. Specifically, Caption offers 19%, 18%, 8%, and 20% higher throughput values for fotonik3d, mcf, roms, and cactuBSSN, respectively, than the best static allocation policy (*i.e.*, 100% or 50% allocation to DDR memory),

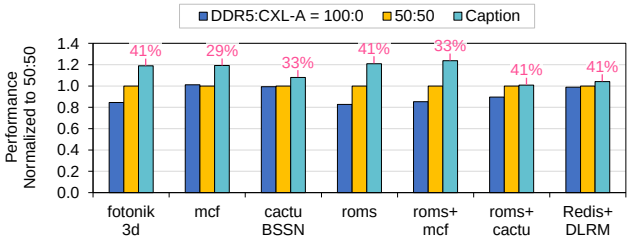


Figure 13: Throughput of each evaluated benchmark or mix, normalized to that with the default static policy allocating 50% of pages to CXL memory. A number atop each bar is the percentage of pages allocated to CXL memory by Caption.

allocating 29%–41% of pages to CXL memory in a steady state. For the mixes of mcf and roms, cactuBSSN and roms, and Redis and DLRM, Caption provides 24%, 1%, and 4% higher throughput values than the best static allocation policy, allocating 33%–41% of pages to CXL memory. Since DLRM and Redis use different throughput metrics, we show a geometric mean value of normalized throughput values of DLRM and Redis as a single throughput value. These demonstrate that Caption captures the immense memory pressure from co-running applications and tunes to the percentage of pages to CXL memory that yields higher throughput than the static allocation policies.

In Figure 13, we do not compare Caption with the static allocation policy for DLRM and Redis individually. This is because we have derived the estimation model after running DLRM (§6.1) and demonstrated that allocating all the pages to DDR memory is best for Redis (§5.2). However, our evaluation shows that Caption presents 80% higher and 4% lower throughput values than allocating 100% and 50% of pages to DDR memory, respectively, for DLRM. For Redis, Caption is able to identify that allocating more memory to low-latency DDR memory is beneficial, and thus offers 3.2% higher throughput than allocating 50% of pages to DDR memory but 8.6% lower throughput than allocating 100% of pages to DDR memory. Albeit not perfect, Caption demonstrates its capability of searching near-optimal percentage values of pages allocated to CXL memory without any guidance from users and/or applications for several workloads with notably different characteristics. Lastly, one of our primary goals is to emphasize the need for a dynamic page allocation policy and show a potential of such a policy. Hence, we leave further enhancement of Caption as future work.

7 RELATED WORK

With the rapid development of memory technologies, diverse heterogeneous memory devices have been introduced. These memory devices are often different from the standard DDR-based DRAM devices, and each memory device offers unique characteristics and trade-offs. These include but are not limited to persistent memory, such as Intel Optane DIMM [84, 88, 90], remote/disaggregated memory [12, 21, 30, 46, 54, 60, 76], and even byte-addressable SSD [1, 8]. These heterogeneous memory devices in the memory hierarchy of datacenters have been applied to diverse domains of applications. For example, in a tiered memory/file system, pages can be dynamically placed, cached, and migrated across different

memory devices, based on their hotness and persistency requirements [2, 5, 14, 22, 23, 31, 48, 49, 51, 61, 64, 68, 78, 79, 81, 82, 89, 97]. Besides, database or key-value store can leverage these memory devices for faster and more scalable data organization and retrieval [10, 16, 17, 45, 47, 53, 92, 95, 98]. Solutions similar to `Caption` were proposed in the context of HBM [18] and storage system [87]. While they have been extensively profiled and studied, CXL memory, as a new member in the memory tier, still has unclear performance characteristics and indications, especially its interaction with CPUs. This leads to new challenges and opportunities for applying CXL memory to the aforementioned domains. This paper aims to bridge the gap of CXL memory understanding, and thus enable the wide adoption of CXL memory in the community. Lastly, our `Caption` is specifically optimized for CXL memory, making the most out of the memory bandwidth available for a given system.

Since the inception of the concept in 2019, CXL has been heavily discussed and invested by researchers and practitioners. For instance, Meta envisioned using CXL memory for memory tiering and swapping [64, 86]; Microsoft built a CXL memory prototype system for memory disaggregation exploration [11, 59]. Most of them used NUMA servers to emulate the behavior of CXL memory. There are also efforts in building software-based CXL simulators [91, 96]. Gouk *et al.* built a CXL memory prototype on FPGA-based RISC-V CPU [29]. There are also a body of work focusing on certain particular applications [52, 71, 74]. Different from the prior studies, this paper presents the first comprehensive study on true CXL memory and compares it with emulated CXL memory using the commercial high-performance CPU and CXL devices with both microbenchmarks and widely-used applications, which can better help the design space exploration of both CXL-memory based software systems and simulators.

8 CONCLUSION

In this paper, we have taken a first step to analyze the device-specific characteristics of true CXL memory and compared them with NUMA-based emulations, a common practice in CXL research. Our analysis revealed key differences between emulated and true CXL memory, with important performance implications. Our analysis also identified opportunities to effectively use CXL memory as a memory bandwidth expander for memory-bandwidth-intensive applications, which leads to the development of a CXL-memory-aware dynamic page allocation policy and demonstrated its efficacy.

ACKNOWLEDGMENTS

We would like to thank Robert Blankenship, Miao Cai, Bhushan Chitlur, Pekon Gupta, David Koufaty, Chidamber Kulkami, Henry Peng, Andy Rudoff, Deshanand Singh, and Alexander Yu. This work was supported in part by grants from Samsung Electronics, PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and NRF funded by the Korean Government MSIT (NRF-2018R1A5A1059921). Nam Sung Kim has a financial interest in Samsung Electronics and NeuroRealityVision.

A ARTIFACT APPENDIX

A.1 Abstract

This appendix explains how to reproduce the results presented in this paper, which is based on an evaluation platform described in Table 1. In the following sections, we provide guidelines to access, build, and setup the environments for individual (micro)benchmarks evaluated in this work. We also provide a set of scripts that automatically collect the results after running experiments. The artifact is available publicly through an archived repository.

A.2 Artifact check-list

- **Program:**
  - Intel MLC (Memory Latency Checker) v3.10 [42]: Figure 3, 4.
  - `memo`: Figure 3, 4.
  - `YCSB` (Yahoo! Cloud Serving Benchmark) [19]: Figure 6, 9, 13.
  - `Redis` [69]: Figure 6, 9, 13.
  - `DSB` (DeathStarBench) [28]: Figure 6.
  - `FIO` (Flexible I/O Tester) [7]: Figure 8.
  - `MERCI` [58]: Figure 9, 11, 12.
  - `SPEC CPU2017` [13]: Figure 12, 13.
  - `Caption`: Figure 12, 13.
- **Compilation:** GCC-11.4.0 (`memo`, FIO, SPEC2017, MERCI), Maven 3 (YCSB). For the other benchmarks, please refer to their repository for proper environemnet setup.
- **Binary:** Intel MLC v3.10, Redis, DeathStarBench.
- **OS environment:** Ubuntu 22.04.2 LTS with different kernel versions and patches.
  - **TPP-related tests:** Linux kernel v5.13 with TPP patch [63].
  - **`Caption`, memory interleaving tests:** Linux kernel v5.19 with m-n memory interleave patch [85].
  - **Else:** Linux kernel v6.2.
- **Hardware:** An evaluation platform with dual-socket Intel Xeon 6430 CPUs, 8-channel DDR5-4800 modules for socket 0, 1-channel DDR5-4800 module for socket 0, and three CXL devices (Table 1).
- **Run-time state:** Fixed CPU frequency (2.1 GHz), Turbo Boost/Hyper-Threading off.
- **Metrics:** The artifact reports memory access latency (ns), memory bandwidth (GB/s), or application throughput (query per second for Redis, inference per second for DLRM).
- **Output:** `.txt` files including collected metrics corresponding to individual experiments.
- **Experiments reproduced:** Figure 3, 4, 6, 8, 9, 13.
- **How much disk space required (approximately)?:** 20 GB storage for DeathStarBench docker images, 50 GB storage for FIO, 5 GB for MERCI dataset.
- **How much time is needed to prepare workflow (approximately)?:** More than 3 hours.
- **How much time is needed to complete experiments (approximately)?:** More than 48 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GPL v2.0.
- **Archived:** https://zenodo.org/record/8332543

A.3 Description

A.3.1 How to access. Our own microbenchmark (`memo`) and proposed page allocation software (`Caption`) are archived at Github and Zenodo. The other (micro)beanchmarks (except for SPEC CPU2017) are open-sourced that are publicly available at:

- Intel MLC v3.10: https://www.intel.com/content/www/us/en/download/736633/763324/intel-memory-latency-checker-intel-mlc.html
- YCSB: https://github.com/brianfrankcooper/YCSB
- Redis: https://github.com/redis/redis
- DSB: https://github.com/delimitrou/DeathStarBench
- FIO: https://github.com/axboe/fio
- MERCI: https://github.com/SNU-ARC/MERCI

A.3.2 Hardware dependencies. We conduct evaluations on an Intel SPR platform equipped with three types of CXL memory devices, as described in Table 1. Sub-NUMA clustering mode (SNC) is disabled for Intel MLC and memo and enabled (*i.e.*, SNC-4) for the later experiments. For a fair comparison with the local CXL memory devices each of which uses a single PCIe link, we limit the remote DDR5 memory to use only a single DRAM channel, as follows:

```
- Socket 0 (Local)
    - DIMM 0 <Inserted>
    - DIMM 1 <Inserted>
    ...
    - DIMM 7 <Inserted>
- Socket 1 (Remote)
    - DIMM 0 <Inserted>
    - DIMM 1 <Not used>
    ...
    - DIMM 7 <Not used>
```

A.3.3 Software dependencies. Our testbed runs Ubuntu 22.04.3 LTS with kernel version 6.2. Most native installation of Ubuntu does not come with libnuma, which is used in memo for allocating memory on a specific NUMA node. The following command will install libnuma on Ubuntu.

```
$ sudo apt install libnuma-dev
```

Additionally, we use cpupower to fix the CPU frequency to 2.1 GHz in all experiments. The following command will install Linux tools including cpupower on Ubuntu.

```
$ sudo apt install linux-tools-$(uname -r)
```

For the rest of the benchmarks, please follow the guidelines in GitHub pages to setup the corresponding environments.

A.4 Installation

To install memo or Caption, we first need to clone the source code:

```
$ git clone https://github.com/ece-fast-lab/cxl_type3_tests
```

Then, go to the source code directory:

```
$ cd memo_ae/src
```

Finally, build the executable binary from the source code:

```
$ make
```

To test Caption, please go to the source code directory for Caption:

```
$ cd caption_ae/src
```

A.5 Experiment workflow

We provide guidelines for BIOS configuration as well as automated scripts for all evaluated workflows. Within these scripts, the common operation goes as follows:

```
$ # set software / hardware environment
$ # run application in loops
$ # unset the environment
```

In most cases, BIOS configuration involves enabling and disabling Sub-NUMA clustering (SNC) and hyperthreading. In the software level, Setting and un-setting the environment involves locking/unlocking the CPU frequency, and creating cgroup directories for fine-grain control over cpu and memory resources. Detailed experiment steps for each experiment (regarding Figure X) is available at the config_figure_X.md file in the folders of the repository.

A.6 Evaluation and expected results

After running each experiment (§A.5), the result data (memory access latency and bandwidth, application throughput, *etc.*) can be automatically collected using a provided script corresponding to each figure. For more detailed instructions, please refer to each run_figure_X.md file in the repository.

A.7 Experiment customization

To run DeathStarBench with the proper NUMA node in Figure 6b, 6c, and 6d, we modify the following docker images to pin their CPU and memory to the desired NUMA node.

- MongoDB: Change entrypoint.sh
- Redis: Install numactl
- Memcached: Install numactl

For Redis and Memcached, the numactl command is embedded in the docker-compose file, whereas MongoDB requires manual changes to the entrypoint.sh within the docker image. Please refer to the setup_docker.md in the repository for the detailed steps in modifying the docker images.

REFERENCES

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[3] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. 2020. Data Direct I/O Characterization for Future I/O System Exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'20)*.

[4] AMD. accessed in 2023. 4th Gen AMD EPYC™ Processor Architecture. https://www.amd.com/en/campaigns/epyc-9004-architecture.

[5] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2020. Assise: Performance and Availability via Client-local NVM in a Distributed File System.

In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[6] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2023. Exploiting CXL-Based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP'22)*.

[7] Jens Axboe. accessed in 2023. Flexible I/O Tester. https://github.com/axboe/fio.

[8] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*.

[9] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. *Noise Reduction in Speech Processing*. Springer.

[10] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment* (2021).

[11] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* (2023).

[12] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'22)*.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE'18)*.

[14] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[15] Daniel W. Chang, Gyungsu Byun, Hoyoung Kim, Minwook Ahn, Soojung Ryu, Nam S. Kim, and Michael Schulte. 2013. Reevaluating the Latency Claims of 3D Stacked Memories. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC'18)*.

[16] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment* (2020).

[17] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[18] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*.

[20] CXL Consortium. accessed in 2023. Compute Express Link (CXL). https://www.computeexpresslink.org.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*.

[22] Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. 2019. HiNUMA: NUMA-Aware Data Placement and Migration in Hybrid Memory Systems. In *Proceedings of the IEEE 37th International Conference on Computer Design (ICCD'19)*.

[23] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[24] eBPF.io. accessed in 2023. eBPF Documentation. https://ebpf.io/what-is-ebpf/.

[25] Samsung Eletronics. accessed in 2023. Scalable Memory Development Kit v1.3. https://github.com/OpenMPDK/SMDK.

[26] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*.

[27] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*.

[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[29] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling with CXL. *IEEE Micro* (2023).

[30] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*.

[31] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. 2018. Reliability-Aware Data Placement for Heterogeneous Memory Architecture. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*.

[32] Rambus Incorporated. accessed in 2023. Memory Interface Chips – CXL Memory Interconnect Initiative. https://www.rambus.com/memory-and-interfaces/cxl-memory-interconnect/.

[33] Intel Corporation. accessed in 2023. 4th Gen Intel Xeon Processor Scalable Family, sapphire rapids. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.

[34] Intel Corporation. accessed in 2023. Difference of Cache Memory Between CPUs for Intel Xeon E5 Processors and Intel Xeon Scalable Processors. https://www.intel.com/content/www/us/en/support/articles/000027820/processors/intel-xeon-processors.html.

[35] Intel Corporation. accessed in 2023. Intel Launches 4th Gen Xeon Scalable Processors, Max Series CPUs. https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-xeon-scalable-processors-max-series-cpus-gpus.html#gs.o28z2f.

[36] Intel Corporation. accessed in 2023. Intel Performance Counter Monitor. https://github.com/intel/pcm.

[37] Intel Corporation. accessed in 2023. Intel Xeon Gold 6430 Processor. https://ark.intel.com/content/www/us/en/ark/products/231737/intel-xeon-gold-6430-processor-60m-cache-2-10-ghz.html.

[38] Intel Corporation. accessed in 2023. Intel® Data Direct I/O (DDIO). https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[39] Intel Corporation. accessed in 2023. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf.

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

[41] Intel Corporation. accessed in 2023. Intel® FPGA Compute Express Link (CXL) IP. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html.

[42] Intel Corporation. accessed in 2023. Intel® Memory Latency Checker v3.10. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[43] Intel Corporation. accessed in 2023. Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors. https://www.intel.com/content/dam/develop/external/us/en/documents/performance-analysis-guide-181827.pdf.

[44] JEDEC – Global Standards for the Microelectronics Industry. accessed in 2023. Main Memory: DDR4 & DDR5 SDRAM. https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram.

[45] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.

[46] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*.

[47] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*.

[48] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[49] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[50] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail

Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[51] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*.

[52] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander. *IEEE Micro* (2023).

[53] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*.

[54] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. 2019. Page Migration Support for Disaggregated Non-Volatile Memories. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'19)*.

[55] Donghyuk Lee, Mike O'Connor, and Niladrish Chatterjee. 2018. Reducing Data Transfer Energy by Exploiting Similarity within a Data Transaction. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*.

[56] Sukhan Lee, Hyunyoon Cho, Young Hoon Son, Yuhwan Ro, Nam Sung Kim, and Jung Ho Ahn. 2018. Leveraging Power-Performance Relationship of Energy-Efficient Modern DRAM Devices. *IEEE Access* (2018).

[57] Sukhan Lee, Kiwon Lee, Minchul Sung, Mohammad Alian, Chankyung Kim, Wooyeong Cho, Reum Oh, Seongil O, Jung Ho Ahn, and Nam Sung Kim. 2018. 3D-Xpath: High-Density Managed DRAM Architecture with Cost-Effective Alternative Paths for Memory Transactions. In *Proceedings of the 27th ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*.

[58] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. 2021. MERCI: Efficient Embedding Reduction on Commodity Hardware via Sub-Query Memoization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[59] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[60] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the ACM/IEEE 36th Annual International Symposium on Computer Architecture (ISCA'09)*.

[61] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[62] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A Manerkar, and Baris Kasikci. 2022. MOESI-prime: Preventing Coherence-Induced Hammering in Commodity Workloads. In *Proceedings of the ACM/IEEE 49th Annual International Symposium on Computer Architecture (ISCA'22)*.

[63] Hasan Al Maruf. accessed in 2023. Transparent Page Placement for Tiered-Memory. https://lore.kernel.org/all/cover.1637778851.git.hasanalmaruf@fb.com/.

[64] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[65] Micron. accessed in 2023. CZ120 memory expansion module. https://www.micron.com/solutions/server/cxl.

[66] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP'15)*.

[67] Montage Technology. accessed in 2023. CXL Memory eXpander Controller (MXC). https://www.montage-tech.com/MXC.

[68] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*.

[69] Redis Ltd. accessed in 2023. Redis. https://redis.io/.

[70] Robert Blankenship. 2020. Compute Express Link (CXL): Memory and Cache Protocols. https://snia.org/sites/default/files/SDC/2020/130-Blankenship-CXL-1.1-Protocol-Extensions.pdf.

[71] Seokhyun Ryu, Sohyun Kim, Jaeyung Jun, Donguk Moon, Kyungsoo Lee, Jungmin Choi, Sunwoong Kim, Hyungsoo Kim, Luke Kim, Won Ha Choi, Moohyeon Nam, Dooyoung Hwang, Hongchan Roh, and Youngpyo Joo. 2023. System Optimization of Data Analytics Platforms using Compute Express Link (CXL) Memory. In *Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp'23)*.

[72] Samsung Semiconductor. accessed in 2023. Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/.

[73] Debendra Das Sharma. 2022. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. *IEEE Micro* (2022).

[74] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2023. Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications. *IEEE Computer Architecture Letters* (2023).

[75] Tom Simon. 2021. Low Power High Performance PCIe SerDes IP for Samsung Silicon - SemiWiki. https://semiwiki.com/events/305345-low-power-high-performance-pcie-serdes-ip-for-samsung-silicon/.

[76] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'20)*.

[77] SK hynix Inc. 2022. SK hynix Introduces Industry's First CXL-based Computational Memory Solution (CMS) at the OCP Global Summit. https://news.skhynix.com/sk-hynix-introduces-industrys-first-cxl-based-cms-at-the-ocp-global-summit/.

[78] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. 2023. Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*.

[79] Kshitij Sudan, Karthick Rajamani, Wei Huang, and John B. Carter. 2012. Tiered Memory: An Iso-Power Memory Architecture to Address the Memory Power Wall. *IEEE Trans. Comput.* (2012).

[80] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *Proceedings of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*.

[81] Majed Valad Beigi, Bahareh Pourshirazi, Gokhan Memik, and Zhichun Zhu. 2020. DeepSwapper: A Deep Learning Based Page Swap Management Scheme for Hybrid Memory Systems. In *Proceedings of the 29th ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'20)*.

[82] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2020. Hybrid2: Combining Caching and Migration in Hybrid Memory Systems. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*.

[83] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. In *Proceedings of the ACM/SPEC on International Conference on Performance Engineering (ICPE'22)*.

[84] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*.

[85] Johannes Weiner. accessed in 2023. [PATCH] mm: mempolicy: N:M interleave policy for tiered memory nodes. https://lore.kernel.org/linux-mm/YqD0%2FtzFwXvJ1gK6@cmpxchg.org/T/.

[86] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*.

[87] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*.

[88] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)*.

[89] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[90] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*.

[91] Yiwei Yang, Pooneh Safayenikoo, Jiacheng Ma, Tanvir Ahmed Khan, and Andrew Quinn. 2023. CXLMemSim: A pure software simulated CXL.mem for performance characterization. *arXiv preprint arXiv:2303.06153* (2023).

[92] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceeings of the USENIX Annual Technical Conference (ATC'20)*.

[93] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *Proceedings of the IEEE/ACM 48th International Symposium on Computer Architecture (ISCA'21)*.

[94] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. 2022. FAERY: An FPGA-accelerated Embedding-based Retrieval System. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*.

[95] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the 26th European Conference on Computer Systems (EuroSys'21)*.

[96] Xu Zhang. accessed in 2023. gem5-CXL. https://github.com/zxhero/gem5-CXL.

[97] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.

[98] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proceedings of VLDB Endowment* (2019).