FEATURE ARTICLE

Improving Key-Value Cache Performance With Heterogeneous Memory Tiering: A Case Study of Compute-Express-Link-Based Memory Expansion

KyungSoo Lee, Sohyun Kim, Joohee Lee, Donguk Moon, Rakie Kim, Honggyu Kim, Hyeongtak Ji, Yunjeong Mun, and Youngpyo Joo, *SK Hynix Inc, Icheon, 17336, South Korea*

Compute Express Link (CXL) memory brings extra bandwidth and capacity via Peripheral-Component-Interconnect-Express-based memory expansion beyond double-data-rate-based dynamic random-access memory. This article introduces the CXL 2.0 memory expansion solution, which incorporates two parts: 1) a CXL memory expander prototype and 2) the heterogeneous memory software development kit. We demonstrate the feasibility of our CXL memory solution by implementing it on CacheLib, Meta's general-purpose key-value caching engine. We highlight how our application design and guidelines for CXL memory enable resolving the shortcomings of conventional memory system architectures. Our proposals enable 1) expanding memory bandwidth and capacity or 2) considerable DRAM savings. Evaluation results show that we can achieve a 25% increase in memory bandwidth, up to 15% throughput gain, and a 9% latency reduction. Furthermore, in hybrid cache using nonvolatile memory (NVM), expanding the RAM cache area with CXL memory, which is relatively cheaper than DRAM, enhances the throughput and hit ratio due to reduced NVM input–output.

Emerging AI and data analytics applications are memory intensive, requiring significantly higher memory bandwidth and capacity than ever before.[1,2,3] Unfortunately, the conventional dynamic random-access memory (DRAM)-only memory system architecture is insufficient in addressing such demands for various reasons, including memory-scaling constraints expected for future double data rate (DDR) interfaces.[4] Recently, the advent of Compute Express Link (CXL) has enabled memory expansion via CXL-compatible memory expanders,[5,6] resulting in the heterogeneous memory system architecture with DRAM and CXL memory that can accommodate the increasing demands for higher memory bandwidth and capacity.

This article presents our investigation into the practicality of applying CXL memory to real-world caching systems, in which key-value caches are typically deployed to cache data from the back-end database-based disk, as illustrated in Figure 1. To improve slow database performance, the key-value cache provides 1) high performance and 2) a high hit ratio. Generally, DRAM is used as the cache storage device for the key-value cache for high performance. The hit ratio is a key metric for key-value caches, defined as the number of cache hits relative to the number of data requests. Upon a cache miss, the application issues a data request to the back-end database. When the application revives the data from the back end, the data are stored in the key-value cache. The cost of responding to a cache miss is very high.

Caching systems span a variety of types, including content delivery network (CDN) caches, key-value caches, and social-graph caches. Irrespective of their specific usage, these systems predominantly rely on DRAM to achieve optimal performance. However, the proliferation of data center caching systems supporting memory-intensive workloads for high performance and a high hit ratio has urged an evaluation of memory system architectures supporting memory expansion beyond DRAM. In this article, we focus on assessing the heterogeneous memory systems supporting DRAM and CXL memory.

0272-1732 © 2024 IEEE
Digital Object Identifier 10.1109/MM.2024.3358861
Date of publication 26 January 2024; date of current version 29 April 2025.

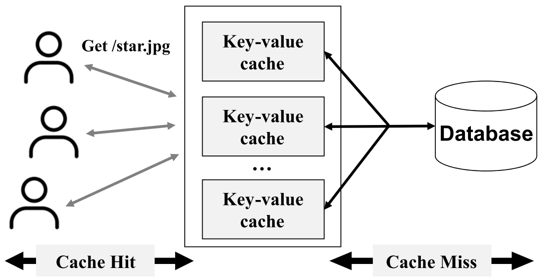


FIGURE 1. Typical key-value cache placement in Web-scale services. Users request data, *star.jpg*. If *star.jpg* is not found in the cache, then the cache will request *star.jpg* from the back-end database and keep it for future use.

We present two distinct memory system architectures for key-value caches: transparent tiering and data tiering. In transparent tiering, DRAM and CXL memory coexist as a unified memory tier, while data tiering differentiates DRAM and CXL memory as separate memory tiers. In particular, we apply these memory system architectures to optimize CacheLib, an open source in-process caching engine developed by Meta.[7]

When new memory devices, such as storage class memory and high-bandwidth memory, emerged, new APIs and programming models, such as Persistent Memory Development Kit[8] and Memkind,[9] were successively developed to facilitate their use. We developed the heterogeneous memory software development kit (HMSDK)[10] for CXL memory, offering developers two programming models: operating system (OS)-level control and application-level control. With OS-level control, the OS manages and allocates DRAM and CXL memory without necessitating application modifications. Meanwhile, application-level control allows users to fine-tune memory usage by manually specifying DRAM and CXL memory allocations separately.

The primary contributions of this article can be summarized as follows:

› The development of near-commercial-performance CXL memory expander prototypes and HMSDK, which offers two programming models: 1) OS-level control and 2) application-level control.
› Detailed enhancements in memory tiering for key-value cache workloads in the data center: 1) high performance and 2) a high hit ratio.
› The introduction of two distinct memory optimization approaches tailored to key-value caches, 1) transparent tiering and 2) data tiering, accompanied by guidelines to help select the most suitable approach based on workload characteristics.
› A comprehensive performance analysis of key-value caching improved by CXL memory expansion, utilizing diverse real-world workloads commonly encountered in data center environments.

With the proposed CXL 2.0 memory expander solution, the overall system bandwidth can be increased, leading to a maximum 15% improvement in throughput performance and a maximum 9% reduction in latency. When compared in terms of capacity, DRAM and CXL memory with the same capacity have identical throughput values and similar hit ratios, indicating that CXL memory is a viable alternative to DRAM. In addition, by adopting CXL memory of larger capacity, the RAM cache area increases, resulting in a higher throughput and hit ratio.

BACKGROUND

CXL

CXL is an interconnect standard built on the Peripheral Component Interconnect Express (PCIe) physical interface and designed to facilitate high-speed and low-latency communication between a CPU and peripheral devices, such as memory, storage, or GPUs. The CXL standard defines three types of devices (type 1, type 2, and type 3) based on their capabilities and use cases. For our specific use case, we focus on type 3 CXL devices, which are specialized for memory expansion. These devices communicate using two subprotocols: CXL.io and CXL.mem. The CXL.io protocol is responsible for the initial discovery, connection, configuration, and ongoing management of the CXL device when linked to a CXL-compatible host. Meanwhile, the CXL.mem protocol enables the processor to access the memory of the CXL device in a byte-addressable manner, meaning that individual bytes can be read or written directly via load or store instructions.[11]

CacheLib

CacheLib is a versatile caching engine developed by Meta, drawing upon extensive experience with various caching scenarios.[7,12] When an object is requested and is not already present in CacheLib, it is retrieved from the back-end database and stored in the cache for future use. Consequently, the hit ratio serves as a critical performance metric for CacheLib. According to Meta,[7] systems employing CacheLib achieve hit ratios ranging from 60% to 90% and are capable of processing up to 1 million requests per second on a single server.

CacheLib provides two caching modes tailored to different application requirements: RAM cache and hybrid cache. In RAM cache mode for high performance, only DRAM is used for cache storage, providing rapid response times for high-bandwidth workloads. On the other hand, hybrid cache expands the storage area beyond DRAM to include nonvolatile memory (NVM), thereby increasing the cache capacity and hit ratio at a relatively lower cost. Figure 2 illustrates the data flow of the hybrid cache for allocate and find operations. Data evicted from the DRAM cache will be inserted into the NVM cache, and data from the NVM cache will be inserted back into the DRAM cache upon lookup.

CacheBench

CacheBench is a benchmark and stress-testing tool designed to evaluate the performance of CacheLib under diverse, real-world workloads commonly encountered in Meta's caching services.[13] The tool comes with a modular request generator that simulates various aspects of real workloads, including the popularity of specific items, keys, and object sizes. CacheBench offers a suite of predefined workloads encompassing social graphs, CDNs, and key-value storage.[14]

CXL 2.0 MEMORY EXPANSION SOLUTION

We introduce our CXL 2.0 memory expansion solution, which comprises two key components: 1) the CXL memory expander prototype and 2) HMSDK.

CXL Memory Prototype

We developed an Enterprise and Data Center Standard Form Factor E3.S 96-GB CXL memory prototype, as shown in Figure 3(a).[15] The prototype is a CXL 2.0-compliant type 3 device, which supports CXL.mem and CXL.io, and features a PCIe Gen 5.0 x8 interface with a maximum bidirectional bandwidth of 32 GB/s.

We measured the bandwidth and latency characteristics of the prototype using the Intel Memory Latency Checker,[16] and the results are summarized in Figure 3(c). Due to the additional delay caused by

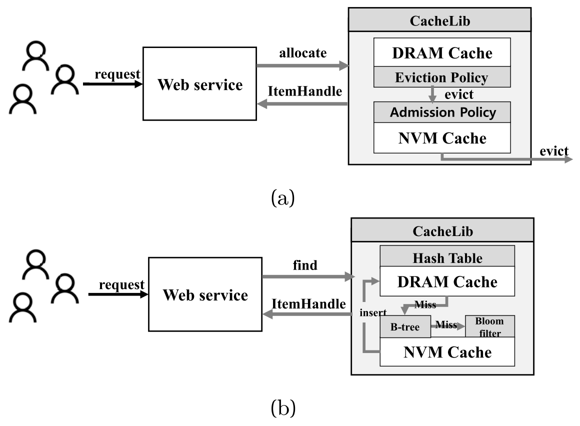
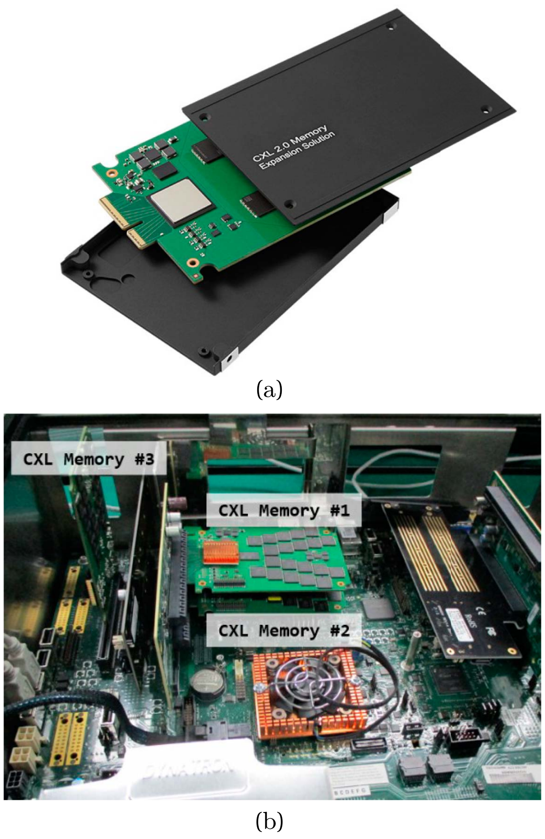


FIGURE 2. Hybrid cache's data flow: (a) allocate and (b) find. DRAM: dynamic random-access memory; NVM: nonvolatile memory.



		CXL(x8 PCIe 5.)	DDR5 (8ch)
Latency		262.5ns	116.7ns
Bandwidth	All Read	17.8 GB/s	219.5 GB/s
	All Write	17.7GB/s	192.2 GB/s
	R:W 3:1	23.6 GB/s	227.5 GB/s
	R:W 2:1	25.1 GB/s	228.1 GB/s
	R:W 1:1	27.2 GB/s	229.4 GB/s

(c)

FIGURE 3. CXL 2.0 memory expander solution. (a) Enterprise and Data Center Standard Form Factor E3.S CXL memory expander prototype. (b) Multi-CXL memory test server. (c) CXL memory expander and DDR5 DRAM performance measured with the Intel Memory Latency Checker. ch: channel; CXL: Compute Express Link; PCIe: Peripheral Component Interconnect Express.

PCIe's serial communication and the CXL memory controller, the prototype's latency is higher than that of local DRAM. Since PCIe supports bidirectional communication, the memory bandwidth for mixed read/write traffic is greater than for read-only traffic. One notable observation is that the bidirectional bandwidth of a single CXL memory module is comparable to the bandwidth of one DDR channel. In other words, a CPU with eight-channel DRAM and four CXL memory expanders may possibly reach around 1.5 times the memory bandwidth of a processor equipped with eight-channel DRAM. HMSDK's bandwidth-aware page interleaving, which is explained in the following section, is based on this observation.

HMSDK

The Linux kernel recognizes each CXL memory module as a separate memory-only nonuniform memory access (NUMA) node.[17,18] To effectively manage memory allocation across NUMA nodes, including CXL memory-only NUMA nodes, we developed HMSDK. HMSDK offers two distinct programming models and supports a novel kernel memory allocation policy.

Programming Models

HMSDK[10] provides application developers with two ways to utilize CXL memory: 1) OS-level control and 2) application-level control. OS-level control allows developers to quickly deploy CXL memory without modifying their application by relying on the OS. On the other hand, application-level control allows developers to manage CXL memory directly within their applications, offering a greater degree of customization.

In OS-level control, memory usage information, such as the NUMA allocation policy and associated parameters, is provided to the underlying HMSDK kernel through environment variables or HMSDK numactl tool arguments. The HMSDK kernel then allocates memory across multiple NUMA nodes based on this information. As illustrated in Figure 4(a), there are two ways—1) setting LD_PRELOAD and 2) using HMSDK numactl—to utilize HMSDK's OS-level control. First, setting LD_PRELOAD, which allows the preloading of designated libraries before application initialization, to the HMSDK library enables it to hook standard libc functions (e.g., mmap, malloc, and free). These hooked functions relay memory usage information defined in system environment variables to the HMSDK kernel. Second, users can employ the HMSDK numactl utility to directly pass memory usage information as arguments to the underlying HMSDK kernel, which then handles memory allocation based on the provided information.

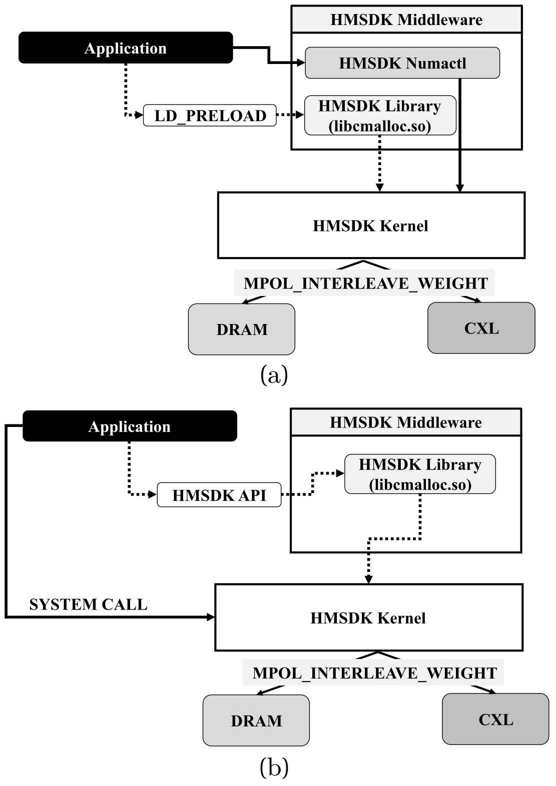


FIGURE 4. HMSDK's programming models. (a) Operating-system-level control. (b) Application-level control. API: application programming interface; HMSDK: heterogeneous memory software development kit.

On the other hand, application-level control allows developers to select which NUMA nodes to allocate memory to by calling functions defined in the HMSDK library. This enables fine-grained optimization that considers characteristics of data structures and places them in appropriate NUMA nodes. Specifically, HMSDK offers 1) the HMSDK API, and 2) the HMSDK system call, as illustrated in Figure 4(b). The HMSDK API (e.g., cxl_malloc, cxl_free, cxl_mmap, etc.) is an interface similar to libc API, and functions defined in the HMSDK library are called via the interface. Meanwhile, the HMSDK system call bypasses the HMSDK library and passes memory usage information directly to the kernel.

New Memory Policy

The CXL memory in KMEM direct access (DAX)[18] is available to the application just like regular memory from the Linux kernel v5.1. The Linux kernel creates CXL memory as a memory-only NUMA. The system memory becomes a multinode consisting of DRAM and CXL memory. MPOL_INTERLEAVE[19] is a traditional memory policy that supports simultaneous access across NUMA nodes. The MPOL_INTERLEAVE policy distributes page allocations uniformly across user-specified NUMA nodes, following their numeric node ID order. This results in an even distribution of page allocations in the system configuration with a DRAM NUMA node and two CXL NUMA nodes (i.e., a ratio of 1:1:1). Unfortunately, this policy does not provide a mechanism for users to adjust the weights between different NUMA nodes (see Figure 5).

The HMSDK kernel offers a new NUMA memory policy, MPOL_INTERLEAVE_WEIGHT, that interleaves memory page allocation across NUMA nodes according to a user-defined memory allocation ratio. Data are stored into DRAM and CXL memory in 4-KB pages, and the CPU can access both DRAM and CXL memory simultaneously on a page-by-page basis. In general, the memory allocation ratio is determined by the difference in bandwidth between DRAM and CXL memory. Figure 5 shows a comparison of the standard memory allocation policy and a new memory policy introduced by HMSDK.

TRANSPARENT TIERING USES HMSDK'S OS-LEVEL CONTROL TO TREAT DRAM AND CXL MEMORY AS THE SAME MEMORY TIER.

Memory Allocation Flow

The memory allocation request information (memory policy and interleave weights) is transferred to the kernel by two novel system calls, `mrange_node_weight` and `set_mempolicy_node_weight`. While `mrange_node_weight` applies a new memory allocation policy for the memory space corresponding to the interleave weights, `set_mempolicy_node_weight` employs the policy to all memory space in the calling process. The HMSDK kernel manages memory allocation information passed in system calls as a weight_list structure, and, when allocating memory to an application, it refers to the weight_list and allocates it to each memory device on a page-by-page basis. The memory allocation process is depicted in Figure 6. Since the number of entries in the weight_list is fewer than the total number of NUMA nodes, the linear scan overhead for each memory allocation request is negligible.

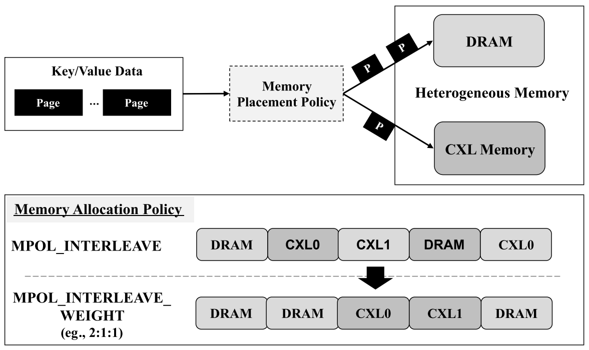


FIGURE 5. MPOL_INTERLEAVE_WEIGHT interleaves pages across NUMA nodes based on user-configurable interleave weights while conventional MPOL_INTERLEAVE interleaves pages evenly. P: page.

Bandwidth Expansion via Page Interleaving

Using MPOL_INTERLEAVE_WEIGHT and the interleave weights value, HMSDK allows page allocations to be stretched across all DRAM and CXL NUMA nodes. Consider, for instance, a single-socket server with eight-channel DDR5-4800 dual inline memory modules (DIMMs) (8*32 GB/s) and four CXL memory devices (4*32 GB/s). Setting a NUMA interleaving ratio of 8:1:1:1:1 results in an equal page allocation distribution over the server's DDR and CXL interfaces, theoretically allowing page interleaving over a total of 12 memory channels. Effective memory bandwidth is expanded by 50% via interleave weights.

MEMORY SYSTEM ARCHITECTURE

We propose two memory system architectures that leverage CXL memory in CacheLib: 1) transparent tiering and 2) data tiering. The proposed architectures use HMSDK to boost memory bandwidth for the RAM cache and memory capacity for the hybrid cache.

Transparent Tiering

Transparent tiering uses HMSDK's OS-level control to treat DRAM and CXL memory as the same memory tier. When the NUMA node allocation ratio is determined based on the interleave weights value, the memory bandwidth of the entire system is expanded from DRAM to DRAM and CXL memory by bandwidth-aware page interleaving. Figure 7(a) shows the process of scaling the memory bandwidth of the system in CacheLib RAM cache mode. The HMSDK numactl executes CacheLib with memory usage information as an

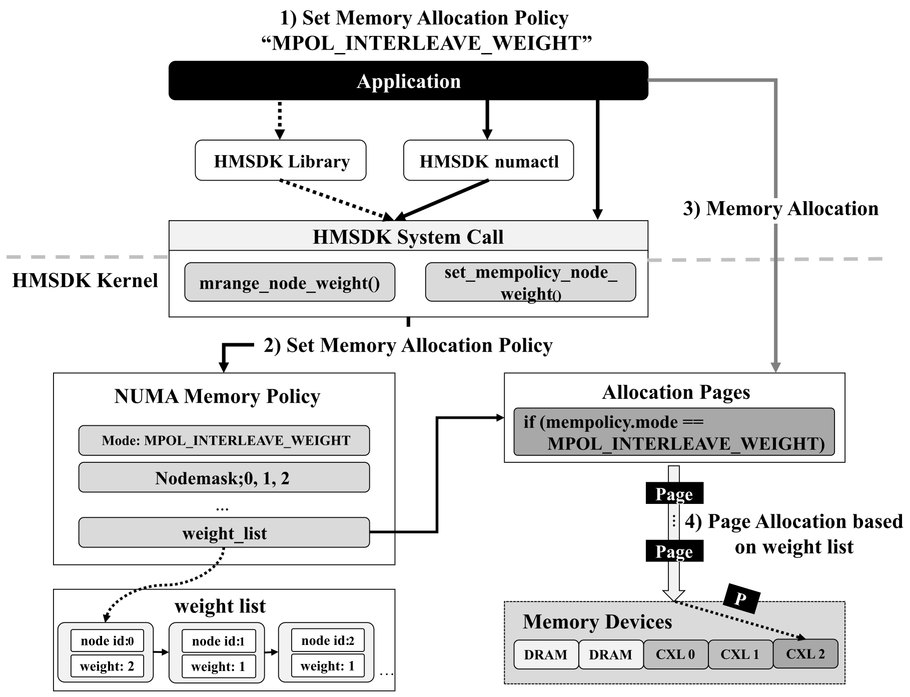


FIGURE 6. HMSDK memory allocation flow.

argument, and all data in CacheLib (e.g., meta, index, key values, etc.) are distributed across DRAM and CXL memory. Transparent tiering is the fastest approach to utilize heterogeneous memory and increase system bandwidth without modifying CacheLib.

Data Tiering

Many users have adopted storage tiering strategies that categorize data as hot, warm, or cold depending on certain factors, such as data type, access frequency, and performance requirements, and keep data in

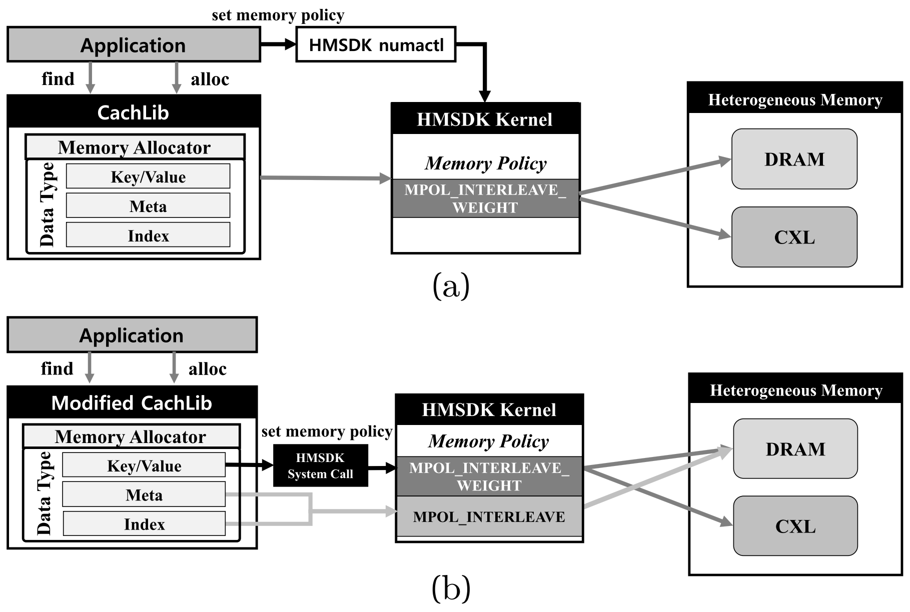


FIGURE 7. Memory system architecture with CXL memory. (a) Transparent tiering. (b) Data tiering.

separate storage areas or media based on data temperature. Data tiering is an architecture that uses HMSDK's application-level controls to tier the memory system and store data based on the distinctive features of each memory device. Figure 7(b) illustrates the data tiering architecture. Data in CacheLib include meta, index, and key values.[7] Meta carries information for retrieving key values in memory upon restart, whereas index saves the location of user data in memory. Typically, indexes are smaller than 4k page size, are latency sensitive, and have many data accesses. Therefore, if the index is allocated in CXL memory, the latency of CXL memory can cause performance degradation in CacheLib. The data tiering architecture addresses this by storing meta and index in DRAM and key values in CXL memory or both, depending on the objects' bandwidth requirements. In addition, the HMSDK system call is used to minimize modifications in CacheLib without adding any additional libraries.

WORKLOAD CHALLENGE

CacheLib operates in a caching system in which user applications and CacheLib instances coexist on a single server. Because multiple CacheLibs run simultaneously, no single CacheLib should dominate the system's memory resources.

To mimic the real-world caching system in our experiments, we examined the workload characteristics listed here:

- *Variability in object size*: Object sizes vary greatly, with little objects being the most prevalent.
- *Memory usage characteristics*: Memory bandwidth and capacity requirements vary depending on the workload.
- *Multi-CacheLib*: Multiple CacheLibs can concurrently be executed on a single server.
- *Hit ratio and throughput*: CacheLib-based systems achieve hit ratios between 60% and 90% and process a million requests per second on a single server.

CDN: Large Objects

The CDN service workloads typically handle large objects of 64 KB or more, requiring high bandwidth and capacity.

To verify the bandwidth expansion needs, we deployed three CacheLib engines in RAM cache mode, each utilizing 20 cores and running on a single-socket server. The cache size was 160 GB, and the hit ratio was 65%. Our results showed that 55% of the CPU and 100% of the eight-channel DDR memory (with a bandwidth of 240 GB/s) were utilized. Further evaluation of a 16-channel DDR memory system demonstrated a bandwidth usage exceeding 300 GB/s. We discovered that certain workloads may be bottlenecked by DRAM bandwidth, which can be resolved by enhancing the system bandwidth with CXL memory.

To validate the capacity expansion needs, we also evaluated the CacheLib engine in hybrid cache mode. The hit ratio was 76%, with a cache size of 540 GB. The results showed that 12% of the eight-channel DDR memory (bandwidth: 30 GB/s) was utilized. By scaling up the capacity with NVM, the cache hit ratio improved, whereas the bandwidth usage did not appear to be excessive, as the access to NVM constrained the overall cache-processing performance. The results of the experiment indicate that a considerable amount of DRAM is necessary to overcome the performance limitations of NVM when the system is input–output (I/O) bound. However, the hybrid cache mode in CacheLib allows the use of CXL memory as a substitute for DRAM due to its extensive cache size, as high DRAM bandwidth is not an essential requirement.

Graph Follower: Small Objects

In contrast to CDN, graph service workloads consist of small objects (10 B to10 KB) and require low bandwidth and capacity. Among the many graph workloads, we used the graph follower workload. For the graph follower workload experiment, we deployed three CacheLib engines in RAM cache mode with a hit ratio of 95% and a cache size of 40 GB. Each engine utilized 20 cores and worked in parallel on a single-socket server. Our results showed that 55% of the CPU and 16% of the DDR memory (with a bandwidth of 40 GB/s) were utilized. Since the DRAM bandwidth is sufficient for graph follower workloads, there is no bandwidth scaling effect of CXL memory to be expected. Also, the large amount of data smaller than 4 KB in graph follower workloads is often not partitioned to DRAM and CXL memory. In this scenario, the system may experience a performance bottleneck due to the low bandwidth (40 GB/s) of CXL memory. However, using CXL memory can save DRAM costs. We analyzed the performance impact of adding CXL memory when the dominant data size is smaller than 4 KB and identified the tradeoffs between the total DRAM capacity and performance for our proposed memory architectures.

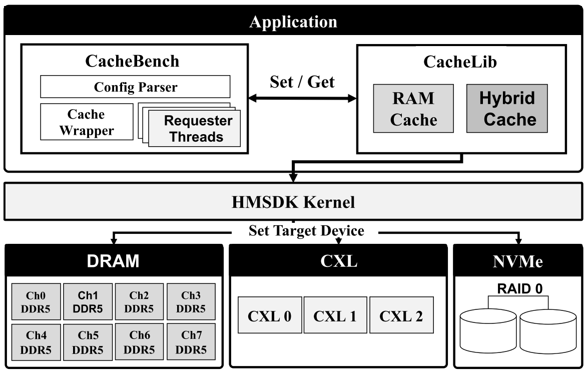


FIGURE 8. Testbed deployment. NVMe: nonvolatile memory express.

EVALUATION ENVIRONMENT

Figure 8 shows a testbed for measuring the performance of CacheLib using CXL memory. Equipped with three CXL memories, the overall system memory bandwidth has improved by about 1.25× compared to a system with only DRAM. Therefore, the bandwidth ratio between DRAM and CXL is 4:1. Hardware and software settings are specified in Table 1. CacheBench was used to evaluate the performance of CacheLib for different memory system architectures, and the OS kernel was switched to the HMSDK kernel. All experiments were conducted when the Meta Caching Services CacheLib requirements of maintaining a hit ratio of 60%–90% for at least 1 million requests per second and of the hit ratio reaching a steady state were met. To exclude the performance impact of the cache-loading process, we used warm restarts,[7] which run CacheLib with all data loaded, and set the size of the hash table as recommended by Meta, which is no more than 50% utilization of the hash table.

TABLE 1. Testbed configuration.*

Hardware specifications
1 × Intel Xeon Platinum 8470 processor (Sapphire Rapids)
52 cores, hyperthreading enabled
16 × 64-GB DDR5-4800, two dual inline memory modules per channel, 1 TB in total
3 × CXL memory expander, 288 GB in total
2 × Platinum P41 NVMe solid-state drive, 2 TB in total
Software specifications
CentOS Stream 8, HMSDK v1.1 (Linux kernel v6.1)
Benchmarks
CacheLib, modified CacheLib for data tiering
CacheBench synthetic CDN, graph follower workload

*CDN: content delivery network; CXL: Compute Express Link; HMSDK: heterogeneous memory software development kit.

BANDWIDTH EXPANSION EVALUATION

This section identifies how the CXL 2.0 memory expansion solution affects throughput and latency performance when implemented in CacheLib's RAM cache mode. We deployed three CacheLib engines on a single server, each using 20 threads. We used two types of workloads: 1) CDN workloads with bandwidth requirements (300 GB/s) exceeding the DRAM maximum bandwidth (240 GB/s) and 2) graph follower workloads with bandwidth requirements (40 GB/s) of much less than the DRAM maximum bandwidth. In experiments with CDN workloads, DRAM-only settings (the baseline) employ 160 GB of DRAM, whereas the transparent tiering and data tiering architectures use 128 GB of DRAM and 32 GB of CXL memory in a 4:1 ratio. With graph follower workloads, DRAM-only settings employ 40 GB of DRAM, whereas transparent tiering and data tiering use 32 GB of DRAM and 8 GB of CXL memory in the same 4:1 ratio.

High-Bandwidth-Demanding Workloads (CDN)

Figure 9 demonstrates the system bandwidth usage of CDN workloads. Thanks to the HMSDK kernel, which allows interleaving across DRAM and CXL memory in a

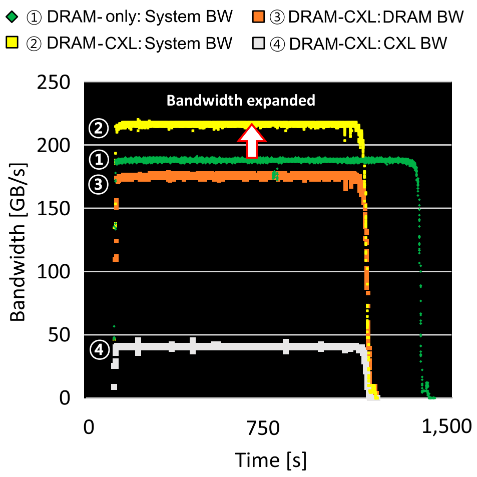


FIGURE 9. System bandwidth usage of CDN workloads. BW: bandwidth.

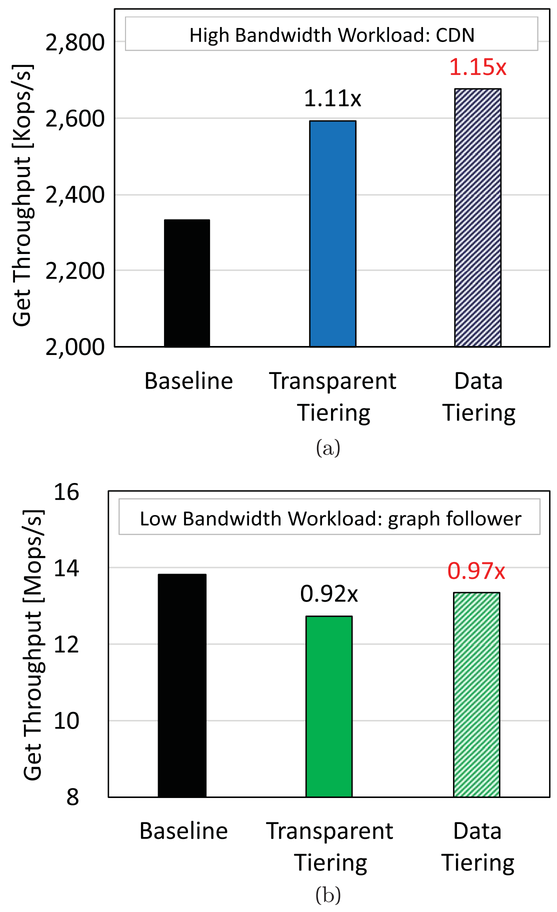


FIGURE 10. Get throughput tests with CXL memory expanders in RAM cache mode. (a) CDN workload (requiring high bandwidth). (b) Graph workload (requiring low bandwidth). CDN: content delivery network.

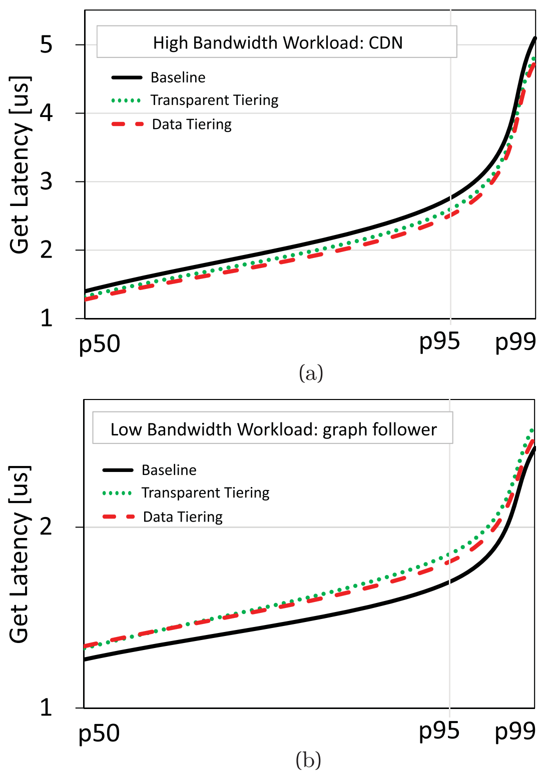


FIGURE 11. Get latency tests with CXL memory expanders in RAM cache mode. (a) CDN workloads (requiring high bandwidth). (b) Graph workloads (requiring low bandwidth).

4:1 ratio, the overall system memory bandwidth has increased. In the case of CDN workloads, as illustrated in Figures 10(a) and 11(a), the system bandwidth expansion resulted in get throughput performance improvement of 11% and 15% in transparent tiering and data tiering, respectively. For p50, p95, and p99 latencies, both transparent tiering and data tiering outperform the baseline, each with a reduction of 5%–6%, and 7%–9%. Results show that selectively pinning some latency-sensitive data in DRAM and performing memory interleaving on the rest of the data bring higher performance gains.

Low-Bandwidth-Demanding Workloads (Graph Follower)

In the case of the graph follower workloads, which do not require system bandwidth expansion, CXL memory with relatively high latency could, rather, lead to performance degradation. The results in Figures 10(b) and 11(b) also show the performance degradation in transparent tiering and data tiering compared to the baseline. However, with data tiering, the performance degradation range is decreased to 3%, which is small enough to compensate for the lower cost of CXL memory over DRAM, leaving room for the use of CXL memory.

CAPACITY EXPANSION EVALUATION

In this section, we conducted experiments to verify the feasibility of employing CXL memory in a hybrid cache mode that utilizes both RAM and NVM for workloads that demand large caches. The CDN service workloads were used, which mainly consist of large objects of 64 KB or greater. In addition, we utilized the data tiering structure, which pins metadata to DRAM and stores key-value data in DRAM or CXL memory of the same capacity. All system configurations include a 500-GB NVMe solid-state drive.

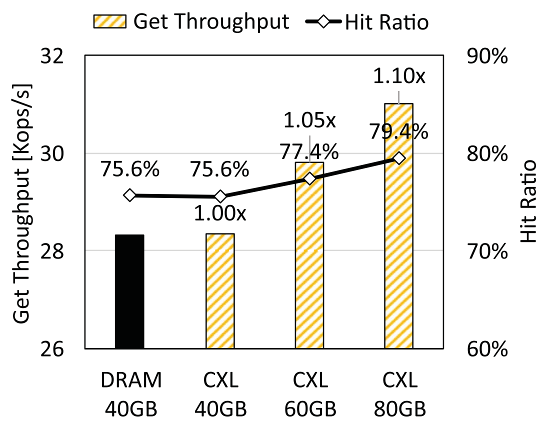


FIGURE 12. CXL memory expanders in hybrid cache mode. Tested with NVMe 500 GB (fixed size) and RAM cache size increasing from 40 to 80 GB. Kops: thousand of operations per second.

High-Capacity Workloads: CDN Workloads

Figure 12 displays the get throughput and hit ratio test results using CXL memory expanders in hybrid cache mode. The results show that both DRAM 40 GB and CXL memory 40 GB have identical get throughput values and similar hit ratio values, indicating that, when the same amount of CXL memory is used instead of DRAM, the same performance is obtained in hybrid cache mode. Furthermore, when configuring hybrid cache with CXL memory 60 GB and CXL memory 80 GB, we can find the effect of improving the get throughput and hit ratio compared to DRAM 40 GB or CXL memory 40 GB. This shows that, by expanding the RAM cache area with CXL memory, which is relatively cheaper than DRAM, we can achieve performance as well as hit ratio improvements owing to reduced I/O. Figure 13 shows block I/O decreasing as the RAM cache expands. Doubling the RAM cache size reduces block I/O by 22% and improves get throughput by 10%. NVMe's slow performance has become a bottleneck in improving overall application performance.

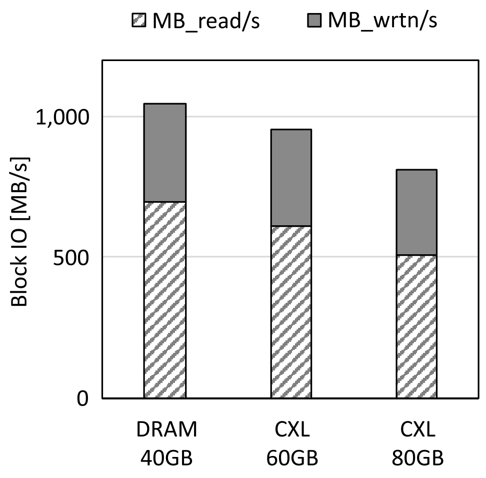


FIGURE 13. Input–output (IO) amount decreases as RAM cache size increases. wtrn: written.

LESSONS LEARNED

Consideration of Memory System Architecture

Our proposed memory system architecture is about how to allocate memory among all (or selected) NUMA nodes. We categorize the memory system architecture type into transparent tiering and data tiering, depending on how the application views CXL memory.

Transparent Tiering (e.g., Using the New Memory Policy in the Linux Kernel and numactl)

In this architecture, DRAM and CXL memory are treated as the same memory tier. Running applications are unaware that they are operating on several NUMA nodes with different properties. This architecture does not require any changes to the application code, but it does require the user to set memory usage information in OS environment variables or as numactl arguments. With transparent tiering, the bandwidth is scaled from DRAM to DRAM and CXL memory for objects larger than 4 KB.

Data Tiering (e.g., Using the HMSDK API and libnuma)

This architecture selectively stores data by tiering DRAM and CXL memory by their properties. Since the application selects which NUMA nodes to allocate memory to, fine-grained optimization based on data characteristics is possible. In this article, we used DRAM to store latency-sensitive index data of CacheLib to further increase the bandwidth scaling effects of CDN workloads. Also, we used both DRAM and CXL memory to store key-value data in both DRAM and CXL memory to demonstrate that CXL memory can replace DRAM when the data require low bandwidth.

Benefits of CXL Memory

CXL memory provides server systems 1) DRAM expansion and 2) DRAM saving.

DRAM Expansion

Since CXL memory does not require DIMM slots, adding four 96-GB CXL memory devices to a single-socket server environment with eight-channel DDR5 32 GB expands 50% of the system bandwidth and 150% of the system capacity. This improves the caching system performance and reduces latency for both DRAM-only and DRAM/NVMe hybrid configurations.

DRAM Saving

When replacing all or part of the DRAM with CXL memory, the performance degradation is either zero or as low as 3% when data tiering is applied, making CXL memory competitive in caching systems that can trade off DRAM capacity for system performance. The DRAM saved by storing data in CXL memory can be utilized for other applications or another CacheLib engine, maximizing the overall system efficiency.

THE ADVENT OF CXL MEMORY PROVIDES AN OPPORTUNITY TO OVERCOME THE DRAM CONSTRAINTS OF CONVENTIONAL MEMORY SYSTEMS AND DRIVE A SHIFT IN MEMORY SYSTEM ARCHITECTURE.

CONCLUSION

Current computer memory system architectures are incapable of meeting scalability requirements beyond DRAM bandwidth or capacity, and this limitation is likely to become a bottleneck for data center services in the near future. The advent of CXL memory provides an opportunity to overcome the DRAM constraints of conventional memory systems and drive a shift in memory system architecture. In this article, we developed a CXL 2.0 memory expansion solution including the E3.S CXL memory prototype and HMSDK. In addition, we proposed two memory system architectures, transparent tiering and data tiering, and applied them to Meta's caching system. By expanding bandwidth and capacity with the suggested structure, we proved that CXL memory can improve the service quality of caching systems that include several bandwidth-intensive or capacity-consuming workloads. Going forward, we will continue to monitor the competitiveness of CXL memory in evolving data center systems and develop optimal CXL memory solutions to enhance memory system architecture.

REFERENCES

1. *Frontswap*. (2021). *The Linux Kernel*. [Online]. Available: https://kernel.org/doc/html/v6.0/mm/frontswap.html
2. J. Ousterhout et al., "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 4, pp. 92–105, 2010, doi: 10.1145/1713254.1713276.
3. J. Gu et al., "Efficient memory disaggregation with infiniswap," in *Proc. 14th USENIX Symp. Networked Syst. Des. Implementation (NSDI)*, 2017, pp. 649–667.
4. M. Natu and T. W. H. Choi, "Questions from the compute express link^{TM} (CXL^{TM}): Supporting persistent memory webinar," Compute Express Link, Beaverton, OR, USA, 2021. [Online]. Available: https://www.computeexpresslink.org/post/questions-from-the-compute-express-link-cxl-supporting-persistent-memory-webinar
5. D. Lee et al., "Elastic use of far memory for in-memory database management systems," in *Proc. 19th Int. Workshop Data Manage. New Hardware*, 2023, pp. 35–43, doi: 10.1145/3592980.3595311.
6. M. Ahn et al., "Enabling CXL memory expansion for in-memory database management systems," in *Proc. 18th Int. Workshop Data Manage. New Hardware*, 2022, pp. 1–5, doi: 10.1145/3533737.3535090.
7. B. Berg et al., "The CacheLib caching engine: Design and experiences at scale," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, USENIX Association, Nov. 2020, pp. 753–768.
8. S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.
9. C. Cantalupo et al., "memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies," Sandia National Lab.(SNL-NM), Albuquerque, NM, USA, Rep. no. SAND2015-1862C 579520, 2015.
10. skhynix. "Heterogeneous memory software development kit." GitHub. Accessed: Feb. 7, 2024. [Online]. Available: https://github.com/skhynix/hmsdk
11. D. D. Sharma, R. Blankenship, and D. S. Berger, "An introduction to the compute express link (CXL) interconnect," 2023, *arXiv:2306.11227*.
12. Cachelib. "Pluggable in-process caching engine to build and scale high performance services." GitHub. Accessed: Feb. 7, 2024. [Online]. Available: https://github.com/facebook/CacheLib
13. "Cachebench: Overview." CacheLib. Accessed: Feb. 7, 2024. [Online]. Available: https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_Overview
14. "Evaluating SSD hardware for Facebook workloads." CacheLib. Accessed: Feb. 7, 2024. [Online]. Available:

https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval/

15. "SK hynix develops DDR5 DRAM CXL^{TM} memory to expand the CXL memory ecosystem." Skhynix. [Online]. Available: https://news.skhynix.com/sk-hynix-develops-ddr5-dram-cxltm-memory-to-expand-the-cxl-memory-ecosystem
16. "Developer zone." Intel. Accessed: Feb. 7, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/ articles/tool/intelr-memory-latency-checker.html
17. Linux. "dax drivers." GitHub. [Online]. Available: http://www.intel.com/content/www/us/en/developer/articles/tool/inteir-memory-latency-checker.com
18. "Memkind support for KMEM DAX option." pmem.io. Accessed: Feb. 7, 2024. [Online]. Available: https://pmem.io/blog/2020/01/memkind-support-for-kmem-dax-option/
19. "NUMA memory policy." Linux. Accessed: Feb. 7, 2024. [Online]. Available: https://www.kernel.org/doc/html/next/admin-guide/mm/numa_memory_policy.html

KYUNGSOO LEE is a senior staff engineer at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include emerging memory system architecture and Compute Express Link (CXL) expansion solutions for data center memory at scale. Lee received his M.S. degree in software engineering from Sogang University. Contact him at kyungsoo3.lee@sk.com.

SOHYUN KIM is a staff engineer at SK Hynix Inc., Icheon, 17336, South Korea. Her research interests include emerging memory system architecture, memory-centric architecture, and CXL expansion solutions for data center memory at scale. Kim received her B.S. degree in computer science and engineering from Sogang University. Contact her at sohyun3.kim@sk.com.

JOOHEE LEE is a staff engineer at SK Hynix Inc., Icheon, 17336, South Korea. Her research interests include emerging memory system architecture and its application to improving data analytics and AI applications. Lee received her M.S. degree in electronic engineering from Sogang University, Seoul, South Korea. Contact her at joohee.lee@sk.com.

DONGUK MOON is the director of the Platform Software Team in Memory System Research at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include enhancing data analytics and AI applications using emerging memory and storage solutions. Moon received his master's degree in electrical engineering from Stanford University and his master's degree in data science from the University of California, Berkeley. Contact him at donguk.moon@sk.com.

RAKIE KIM is a principal engineer at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include developing software developer's kits for the efficient use of CXL memory in data centers. Kim received his M.S. degree in computer science and engineering from Hanyang University. Contact him at rakie.kim@sk.com.

HONGGYU KIM is a senior staff engineer at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include memory profiling and management in Linux kernel. Kim received his M.S. degree in computer science and engineering from Seoul National University. Contact him at honggyu.kim@sk.com.

HYEONGTAK JI is a staff engineer at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include emergency memory system architecture and its management in the Linux Kernel. Ji received his B.S. degree in computer science and engineering from Hanyang University. Contact him at hyeongtak.ji@sk.com.

YUNJEONG MUN is a staff engineer at SK Hynix Inc., Icheon, 17336, South Korea. Her research interests include emerging memory and system software. Mun received her B.S. degree in computer science and engineering from Sogang University. Contact her at yunjeong.mun@sk.com.

YOUNGPYO JOO is a research fellow and leader of the Software Solution Group at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include software solutions that utilize and support next-generation memory solutions, such as computational and pooled memory. Joo received his Ph.D. degree from Seoul National University. Contact him at youngpyo.joo@sk.com@sk.com.