Exploring Performance and Cost Optimization with ASIC-Based CXL Memory

Yupeng Tang*, Ping Zhou§, Wenhui Zhang§, Henry Hu§, Qirui Yang§, Hao Xiang§, Tongping Liu§, Jiaxin Shan§, Ruoyun Huang§, Cheng Zhao§, Cheng Chen§, Hui Zhang§, Fei Liu§, Shuai Zhang§, Xiaoning Ding§, Jianjun Chen§
Yale University* ByteDance§

Abstract

As memory-intensive applications continue to drive the need for advanced architectural solutions, Compute Express Link (CXL) has risen as a promising interconnect technology that enables seamless high-speed, low-latency communication between host processors and various peripheral devices. In this study, we explore the application performance of ASIC CXL memory in various data-center scenarios. We then further explore multiple potential impacts (e.g., throughput, latency, and cost reduction) of employing CXL memory via carefully designed policies and strategies. Our empirical results show the high potential of CXL memory, reveal multiple intriguing observations of CXL memory and contribute to the wide adoption of CXL memory in real-world deployment environments. Based on our benchmarks, we also develop an Abstract Cost Model that can estimate the cost benefit from using CXL memory.

CCS Concepts: • **Software and its engineering → Memory management**; • **Hardware → Memory and dense storage**; • **General and reference → Empirical studies**.

Keywords: Datacenters, Operating Systems, Memory Management, CXL-Memory, measurement

1 Introduction

In an age marked by the surge of memory-intensive applications, such as machine learning tasks and High-Performance Computing (HPC) applications, there is an urgent need for expanding the memory capacity and bandwidth [1–3]. For instance, a machine learning application with 175 B model requires 700 GB of memory to hold its parameters only, not to mention memory requirements for intermediate results and others. That is, the memory requirements of modern applications could easily exceed the memory capability of a single machine due to physical constraints, such as availability of DDR DIMM slots and thermal issues, as well as cost considerations of employing high-density DIMMs [2, 3].

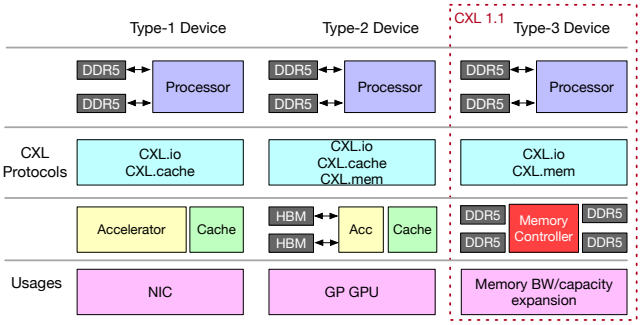


Fig. 1. CXL Overview. In this study, we focus on commercial CXL 1.1 Type-3 devices, leveraging CXL.io and CXL.mem protocols for memory expansion in single-server environments.

To meet such urgent demands, Compute Express Link (CXL) [3–6] is introduced as a groundbreaking interconnect technology. CXL promises significant expansion of memory capacity and bandwidth by attaching external memory devices (e.g., DRAM, Flash or persistent memory) to PCIe slots. Unlike its predecessors, CXL enables a more dynamic and heterogeneous computing environment, leading to various design trade-offs for performance and cost gains. Commercially debuting with version 1.1, CXL allows direct attachment of external memory devices to the host machine, enabling a unified and coherent memory address space. In such configuration, CXL is predominantly used as a way of memory expansion. For example, AsteraLabs' A1000 [7] CXL memory expansion card supports up to 4xDDR5 RDIMMs, enabling up to 2 TB of additional memory for a single server.

Although substantial studies on CXL memory have been performed in the past [3, 5, 6, 8–12], there remains a significant gap of employing these studies to guide the integration of CXL practically. In particular, we observe the following issues: (1) Much of the current literature has focused on evaluating CXL hardware through simulations [6, 8] or using FPGA-based setups [11, 12]. Although a limited number of studies have begun to assess the raw performance of ASIC-based CXL hardware [11, 13], there remains a gap in understanding how different system configurations influence the

*Work done during an internship at ByteDance.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3650061

performance of data center applications using CXL memory. Furthermore, the specific applications that could substantially benefit from CXL memory expansion are not yet fully identified. (2) While existing studies have begun to explore the cost implications of employing CXL technology, such as the work on memory pooling cost models presented in [14], a critical gap remains in understanding the cost-effectiveness of migrating particular types of applications or services to memory expansions facilitated by CXL. (3) Given the restricted availability of CXL ASIC hardware, the research community faces a notable scarcity of open-source empirical data. This limitation hinders efforts to fully comprehend the performance capabilities of such hardware or to develop performance models based on empirical evidence.

Our study aims to fill existing knowledge gaps by conducting detailed evaluations of CXL 1.1 for memory-intensive applications, leading to several *intriguing observations*: Contrary to the common perception that CXL memory, due to its higher latency, should be considered a separate, slower tier of memory [8, 9], **we find that shifting some workloads to CXL memory can significantly enhance performance**, even if local memory's capacity and bandwidth are underutilized. This is because using CXL memory can decrease the overall memory access latency by alleviating bandwidth contention on DDR channels, thereby improving application performance. From our analysis of application performance, we have formulated an abstract cost model (§6) that predicts substantial cost savings in practical deployments.

In summary, the major contributions of this paper are:

- **Empirical Evaluation of ASIC CXL Hardware**: Our study comprehensively examines the performance of ASIC-based CXL hardware and system configurations in data center applications, offering insights on optimizing CXL memory utilization.
- **Cost-Benefit Analysis**: We undertake a comprehensive cost-benefit analysis and develop an Abstract Cost Model to evaluate how CXL memory could substantially reduce real-world applications' TCO (Total Cost of Ownership).
- **Open-source data on CXL ASIC performance**: We open source all data and testing configurations under `https://github.com/bytedance/eurosys24-artifacts`.

The paper organizes as follows. §2 introduces basic information of CXL and environment setup for the evaluations. §3 presents basic performance characteristic of CXL memory expansion. §4 and §5 presents findings and suggestions of using CXL as the expansion of memory capacity and bandwidth on data center workloads. §6 provides a detailed analysis on the potential cost benefits brought by CXL. §7 discusses how our insights are applicable to future generations of CXL. §8 describes related work, and §9 concludes the paper.

2 Background and Methodology

This section presents an overview of CXL technology, followed by our experimental setup and methodologies.

2.1 Compute Express Link (CXL) Overview

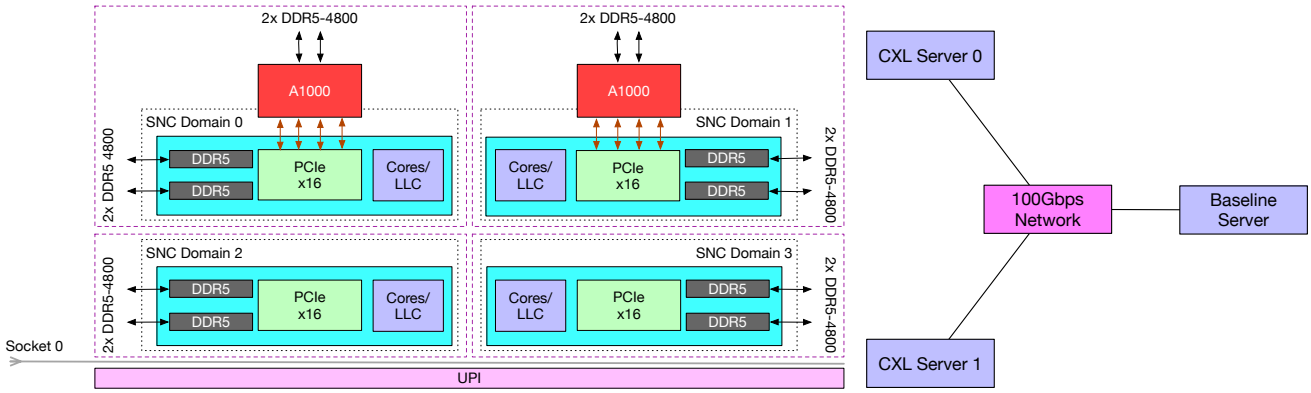
Compute Express Link (CXL) [15] is a standardized interconnect technology that facilitates communication between processors and various devices, including accelerators, memory expansion units, and smart I/O devices. CXL is built upon the physical layer of PCI Express® (PCIe®) 5.0 [16], providing native support for x16, x8, and x4 link widths with data rates of 32.0 GT/s and 64.0 GT/s. The CXL transaction layer is implemented through three protocols: CXL.io, CXL.cache, and CXL.mem, as depicted in Fig. 1. *CXL.io* protocol is based on PCIe 5.0 and handles device discovery, configuration, initialization, I/O virtualization, and direct memory access (DMA). *CXL.cache* enables CXL devices to access the host processor's memory. *CXL.mem* allows the host to access memory attached to devices using load/store commands.

CXL devices are categorized into three types, each associated with specific use cases: (1) *Type-1 devices* like SmartNICs utilize CXL.io and CXL.cache for DDR memory communication. (2) *Type-2 devices*, including GPUs, ASICs, and FPGAs, employ CXL.io, CXL.cache, and CXL.mem to share memory with the processor, enhancing various workloads in the same cache domain. (3) *Type-3 devices* leverage CXL.io and CXL.mem for memory expansion and pooling. This allows for increased DRAM capacity, enhanced memory bandwidth, and the addition of persistent memory without sacrificing DRAM slots. Type-3 devices complement DRAM with CXL-enabled solutions, benefiting high-speed, low-latency storage.

The commercially available version of CXL is 1.1, where a CXL 1.1 device can only serve as a single logical device accessible by one host at a time. Future generations of CXL, like CXL 2.0, are expected to support the partitioning of devices into multiple logical units, enabling up to 16 different hosts to access different portions of memory [17]. In this paper, our focus is on commercially available CXL 1.1 Type-3 devices, specifically addressing single-host memory expansion.

2.2 Hardware Support for CXL

Recent announcements have introduced CXL 1.1 support for Intel Sapphire Rapids processors (SPR) [18] and AMD Zen 4 EPYC "Genoa" and "Bergamo" processors[19]. While commercial CXL memory modules are provided by vendors such as Asteralabs [7], Montage [20], Micron [21], and Samsung [13], CXL memory expanders are predominantly in prototype stages, with only limited samples available, making access difficult for university labs. Consequently, due to the scarcity of CXL hardware, research into CXL memory has largely depended on NUMA-based emulation [8, 9] and FPGA implementations [11, 12], each with inherent limitations:



(a) CXL Server(socket 0 illustrated here). (b) Server setup.

Fig. 2. CXL Experimental Platform. (a) Each CXL server is equipped with two A1000 memory expansion cards. SNC-4(§3.1) is enabled only for the raw performance benchmarks(§3) and bandwidth-bound benchmarks(§5), and each SNC Domain is equipped with two DDR5 channels . (a) illustrates Socket 0; Socket 1 shares a similar setup except for the absence of CXL memory. (b) Our platform comprises two CXL servers and one baseline server. The baseline server replicates the same configuration but lacks any CXL memory cards.

NUMA-based emulation. Given the cache coherent nature and comparable transfer speed of CXL and UPI/xGMI interconnects, NUMA-based emulation [8, 9] is widely adopted to enable fast application performance analysis and software prototyping as the CXL memory is exposed as a remote NUMA node. However, NUMA-based emulation fails to accurately capture the performance characteristics of CXL memory due to differences from CXL and UPI/xGMI interconnects [22], as shown in previous research [11].

FPGA-based implementation. Intel and other hardware vendors use FPGA hardware to implement CXL protocols [23], bypassing the performance inconsistencies of NUMA-based emulation. However, FPGA-based CXL memory falls short in fully utilizing memory chip performance due to its lower operating frequency compared to ASICs [24]. FPGAs prioritize flexibility over performance and are suitable for early-stage CXL memory validation but not production deployment. Intel's recent evaluation [11] uncovered performance issues in FPGA implementations, including reduced memory bandwidth during concurrent thread execution. This hampers rigorous evaluations for memory capacity- and bandwidth-bound applications, which are key use cases for CXL memory expanders. Further discussion on the performance disparity between CXL ASIC and FPGA controllers is in §3.

To the best of our knowledge, we are one of the pioneers in uncovering the performance characteristics of actual ASIC prototypes designed for CXL memory expansion. The ASIC CXL memory controller we have employed is the A1000 [7] developed by AsteraLabs, which implements the CXL interface at speeds of up to 32 GT/s per lane, supporting up to 16 lanes in total. This controller has the capability to accommodate up to 4 DDR5-5600 RDIMM slots, providing a total memory capacity of 2TB.

2.3 Software Support for CXL

While hardware vendors are actively advancing CXL production, a notable deficiency exists in software and OS kernel support for CXL memory. This deficiency has prompted the utilization of specific software enhancements. We summarize the most recent patches in the Linux Kernel that add CXL-aware support, namely: (1) the interleaving policy support (unofficial) and (2) the hot page selection support (official since Linux Kernel v6.1).

N:M Interleave Policy for Tiered Memory Nodes.

Traditional memory interleave policies distribute data evenly across memory banks, often using a 1:1 ratio. However, the advent of tiered memory systems, which feature CPU-less memory nodes with diverse performance traits, demands more nuanced strategies for optimizing memory bandwidth, especially for bandwidth-heavy applications. The interleave patch [25] introduces an innovative N:M interleave policy to address this, allowing for an allocation scheme where N pages are directed to high-performance (top-tier) nodes and M pages to lower-tier nodes. For example, using a 4:1 ratio directs 80% of traffic to top-tier nodes and 20% to low-tier nodes, adjustable through the `vm.numa_tier_interleave` parameter. While the patch showcases compelling evaluation results [25], it's crucial to note that optimal memory distribution depends on specific hardware and application characteristics. Given the higher latency of CXL memory, as demonstrated in §3, performance-sensitive applications should undergo thorough profiling and benchmarking to maximize the advantages of interleaving and mitigate potential performance trade-offs.

NUMA Balancing & Hot Page Selection.

The memory subsystem, now termed a memory tiering system, accommodates various memory types like PMEM and

CXL Memory, each with differing performance characteristics. To optimize system performance, "hot pages" (frequently accessed) should reside in faster memory tiers like DRAM, while "cold pages" (less frequently accessed) should be in slower tiers like CXL memory. Recent Linux Kernel patches address this:

1. The *NUMA-balancing* patch [26] uses a latency-aware page migration strategy, focusing on promoting recently accessed pages (MRU). It scans NUMA balancing page tables and hints page faults. However, it may not accurately identify high-demand pages due to extended scanning intervals, potentially causing latency issues for some workloads.

2. The *Hot Page Selection* patch" [27] introduces a Page Promotion Rate Limit (RPRL) mechanism to control the rate of page promotions and demotions. While this extends promotion/demotion times, it improves workload latency. The hot page threshold is dynamically adjusted to align with the promotion rate limit.

Additionally, research prototypes like TPP [9] share a similar concept with optimizations and are being considered for integration into the Linux Kernel [28]. However, we faced challenges with TPP when running memory-bandwidth-intensive applications, resulting in unexplained performance degradation. Hence, we rely on the well-tested kernel patches integrated into Linux Kernel since version 6.1.

2.4 Experimental Platform Description

The evaluation testbed, as illustrated in Fig. 2(b), consists of three servers. Two of these servers are designated as CXL experiment servers. Each of these servers is equipped with dual Intel Xeon 4th Generation CPUs (Sapphire Rapids, or SPR), 1 TB of 4800 MHz DDR5 memory, two 1.92 TB SSDs, and a pair of A1000 CXL Gen5 x16 ASIC memory expanders modules from AsteraLabs, each with 256 GB of 4800MHz memory (resulting in a total of 512 GB memory per server). Both A1000 memory modules are attached to socket 0. The third server serves as the baseline and is configured identically to the CXL experiment servers, except for the absence of the CXL memory expanders. It is designated for initiating client requests and running workloads that strictly utilize the main memory during the application assessments. All servers are interconnected via 100 Gbps Ethernet links.

3 CXL 1.1 Performance Characteristics

In this section, we assess the performance of the CXL memory expander and compare it directly with main memory, which we designate as **MMEM** for clarity against CXL memory. We analyze workload patterns and evaluate performance differences between local and remote socket scenarios.

3.1 Experimental Configuration

For each dual-channel A1000 ASIC CXL memory expander [7], we connect two DDR5-4800 memory channels, achieving a total capacity of 256 GB. To provide a fair comparison between MMEM and CXL-attached DDR5 memory, we utilize the Sub-NUMA Clustering (SNC) [29] feature to ensure the number of memory channels is the same in both settings.

Sub-NUMA Clustering(SNC). Sub-NUMA Clustering (SNC) serves as an enhancement over the traditional NUMA architecture. It decomposes a single NUMA node into multiple smaller semi-independent sub-nodes (domains). Each sub-NUMA node possesses its own dedicated local memory, L3 caches, and CPU cores. In our experimental setup (Fig. 2(a)), we partition each CPU into four sub-NUMA nodes. Each sub-NUMA node is equipped with two DDR5 memory channels connected to two 64 GB DDR5-4800 DIMMs. Enabling SNC requires setting the IMC (Integrated Memory Controllers) to 1-way interleaving. According to the specifications, a single DDR5-4800 channel has a theoretical peak bandwidth of 38.4 GB/s [6]. Therefore, each sub-NUMA node has a combined memory bandwidth of up to 76.8 GB/s.

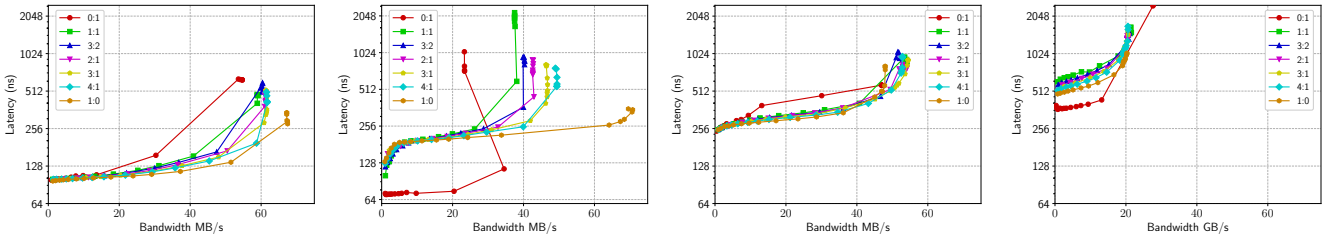
Intel Memory Latency Checker (MLC). We leverage Intel's Memory Latency Checker (MLC) to examine loaded-latency for various read-write workloads, adopting a 64-byte access size same as prior work [11]. We deploy 16 MLC threads, and it's important to note that while the thread count is a configurable parameter in MLC, it doesn't directly dictate memory request concurrency. MLC assigns separate memory segments for each thread to access simultaneously. Specifically, when evaluating loaded latency, MLC incrementally increases the operation rate of each thread. Our findings indicate that employing 16 threads with MLC precisely measures both the idle and loaded latency and the point at which bandwidth becomes saturated. MLC accommodates a broad spectrum of workloads including those with varied read-write mixes and non-temporal writes.

Our study is focused on addressing the following research questions:

- How is the performance of the CXL-attached memory compared to that of local-socket/remote-socket main memory?
- What is the performance impact of the CXL memory under different read-write ratios and access patterns (random vs. sequential)?
- How do main memory and CXL memory behave under high memory load conditions?

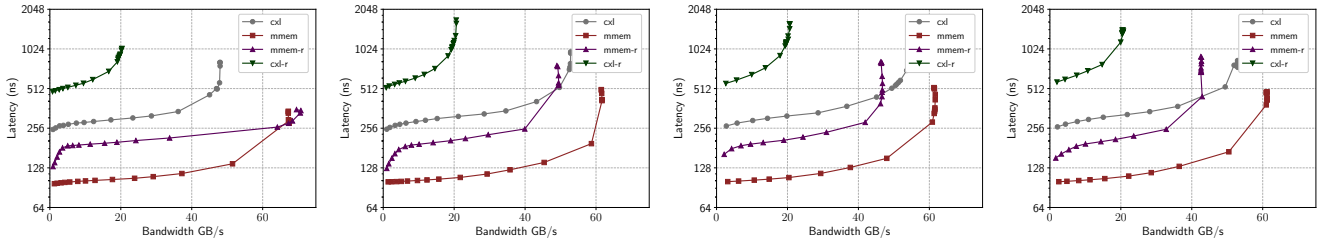
3.2 Basic Latency and Bandwidth Characteristics

This section outlines our findings on memory access latency and bandwidth for different memory configurations: local-socket main memory (MMEM), remote-socket main memory (MMEM-r), CXL memory (CXL), and remote-socket CXL memory (CXL-r). Figure 3(a) shows the loaded latency curve for MMEM under varied read-write mixes. The read-only workload hits a peak bandwidth of roughly 67 GB/s, reaching 87% of its theoretical maximum. Yet, as write operations increase, bandwidth dips, with write-only tasks dropping to 54.6 GB/s. We note an initial memory latency of about 97 ns, which



(a) Local-socket MMEM (b) Remote-socket MMEM (c) Local-socket CXL (d) Remote-socket CXL

Fig. 3. Overall effect of read-write ratio on MMEM and CXL across different distances. The workloads are represented by read:write ratios (e.g., 0:1 for write-only, 1:0 for read-only). Accessing CXL memory locally incurs higher latency compared to MMEM but is more comparable to accessing MMEM on a remote socket. MMEM bandwidth peaks at 67 GB/s, versus 54.6 GB/s for CXL memory. Performance significantly declines when accessing CXL memory on a remote socket (§3.2). In specific scenarios, such as the write-only workload (0:1) in (b), the plot may show instances where bandwidth decreases and latency increases with heavier loads. The Y-axis is on a logarithmic scale.



(a) Read-only workload (b) Read:Write = 4:1 workload (c) Read:Write = 3:1 workload (d) Read:Write = 2:1 workload

(e) Read:Write = 1:1 workload (f) Write-only workload (g) Read-only workload(random) (h) Write-only workload(random)

Fig. 4. A detailed comparison of MMEM versus CXL over diverse NUMA/socket distances and workloads. (a)-(f) shows the latency-bandwidth trend difference of accessing data from different distances in sequential access pattern, sorted by the proportion of write. We refer to main memory as **MMEM**, with MMEM-r and CXL-r representing remote socket MMEM and cxl memory access, respectively. The Y-axis is on a logarithmic scale.

spikes exponentially as bandwidth nears full capacity, a sign of bandwidth contention [30, 31]. Interestingly, latency starts to significantly increase at 75%-83% of bandwidth utilization, surpassing prior estimates of 60% from earlier studies [30].

Figure 3(b) illustrates the latency differences when accessing MMEM via a remote socket. For read-only tasks, latency begins at approximately 130 ns, contrasting sharply with just 71.77 ns for write-only operations. This reduced latency for write-only workloads results from non-temporal writes, which proceed asynchronously without awaiting confirmation. Despite read-only tasks achieving maximum bandwidth comparable to that of local MMEM, incorporating more write operations significantly diminishes bandwidth, attributed to the additional UPI traffic necessitated by cache coherence protocols. Interestingly, the write-only workload generate minimal UPI traffic but suffer the lowest bandwidth as it utilize only one direction of the UPI's bidirectional capabilities. Moreover, latency escalation occurs earlier in remote socket memory accesses than in local ones, primarily due to queue contention at the memory controller.

Fig. 3(c) illustrates the latency curve for CXL memory expansion, demonstrating a minimum latency of 250.42 ns. Interestingly, despite additional PCIe and CXL memory controller overhead on the datapath, accessing CXL follows the same "Bandwidth contention" trend as MMEM. The latency

of accessing CXL on the same socket remains relatively stable as bandwidth increases, with a maximum bandwidth of around 56.7 GB/s, achieved when the workload is 2:1 read-write ratio. The reduction in maximum bandwidth compared to DRAM is attributed to PCIe overhead, such as extra headers. The maximum bandwidth for read-only workloads is smaller due to PCIe bi-directionality, preventing full bandwidth utilization. Fig. 3(d) reveals the latency-bandwidth plot for accessing CXL from a remote socket, incurring an exceptionally high idle latency of 485 ns. In addition, the maximum memory bandwidth is unexpectedly halved, reaching just 20.4 GB/s for 2:1 read-write ratio, which is a much more severe performance drop compared to accessing MMEM from the remote NUMA node in Fig. 3(d). Since running a read-only towards a CXL Type-3 device on the remote socket does not generate substantial coherence traffic, initial speculation regarding cache coherence is ruled out. Further investigation utilizing the Intel Performance Counter Monitor (PCM) [32] also confirms that the UPI utilization is consistently below 30%. Discussions with Intel suggest this performance bottleneck is likely due to limitations in the Remote Snoop Filter (RSF) on the current CPU platform, anticipated to be addressed in the next-generation processors [33].

3.3 Different Read-Write Ratios & Access Pattern

Fig. 4(a)-4(f) present a performance comparison for a specific workload with varying read-write ratios. The results align with our observation that accessing CXL from a remote socket introduces exceptionally high latency and low bandwidth. When accessing CXL from the same socket, latency is 2.4-2.6 × that of local DDR and 1.5-1.92 × that of remote socket DDR. This suggests that running applications directly on CXL may significantly drop performance. However, when workloads span multiple NUMA nodes within the same socket, accessing CXL locally is comparable to accessing remote NUMA node memory. Additionally, the latency-bandwidth knee-point shifts to the left as the proportion of write operations in the workload increases. Fig. 4(g) and 4(h) display the results of running both read-only and write-only workloads, utilizing random access patterns instead of sequential access. Notably, we do not observe any significant performance disparities under these conditions.

3.4 Key insights

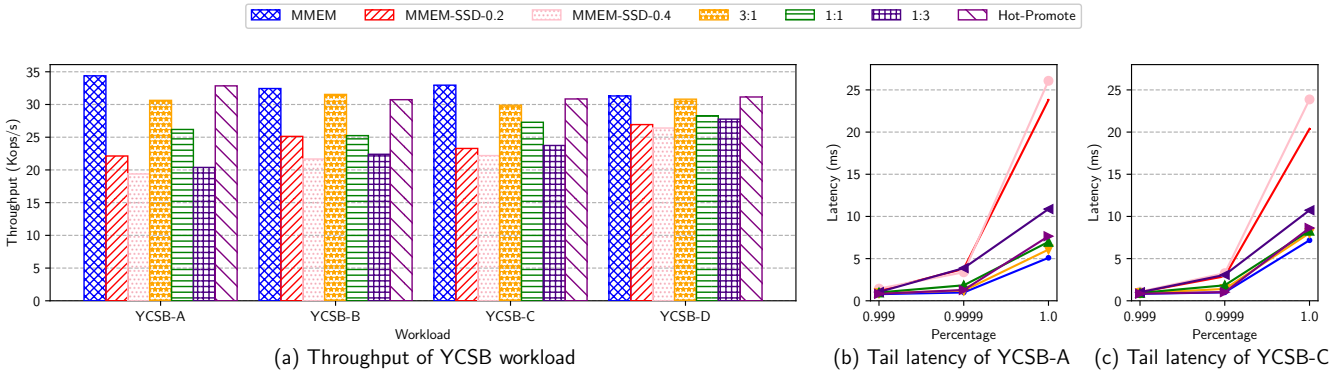
Avoiding Remote Socket CXL Access. CXL memory expansion is commonly utilized for applications that are demanding in terms of memory, particularly those limited by memory capacity or bandwidth. In such contexts, accessing memory across sockets is not uncommon. It is important for software developers to recognize the potential decline in performance when CXL memory is accessed from a remote socket and to strategize against cross-socket CXL memory accesses in their applications. Additionally, hardware vendors should perform cooperative testing and validation of their products to ensure compatibility between CXL memory modules and the processors' CXL support. With adequate support for the CXL 1.1 protocol, we expect that the maximum bandwidth attainable when accessing CXL memory across sockets could approximate the bandwidth seen when accessing MMEM across sockets.

Bandwidth Contention Previous research [6, 31] has brought attention to issues related to bandwidth contention. We further examine how memory latency varies with varying read-write ratios under bandwidth contention. While latency remains relatively stable at low to moderate bandwidth utilization levels, it increases exponentially as bandwidth approaches higher levels, primarily due to queuing delays in the memory controller [30]. Furthermore, the knee-point in latency shifts to lower memory bandwidth when there is a higher proportion of write operations in the workload. Interestingly, CXL-attached memory has often been characterized by industry and research community as 'tiered memory' [11, 25, 28], suggesting that it serves as a slower and less performant memory layer to be considered only when MMEM is fully utilized. However, we argue against this simplistic view of CXL-memory. Allocators and kernel-level page placement policies should consider the available bandwidth in MMEM. Even if a substantial portion of memory bandwidth in MMEM remains unused, e.g., 30%, offloading a portion of the workload, e.g., 20%, to CXL memory can lead to overall performance improvements. Our recommendation is to regard CXL memory as a valuable resource for load balancing, even when local DRAM bandwidth is not fully utilized. Subsequent real-world evaluations support these insights (§5).

Comparison with FPGA-based CXL implementations. Intel recently disclosed latency and bandwidth performance metrics for their FPGA-based CXL prototype [11]. While they provided insights into relative latency and bandwidth efficiency for soft and hard IP implementations, performance under load was not shared. Our measurements indicate that the ASIC CXL solution only introduces a less than $2.5x$ overhead in access latency compared to MMEM, surpassing most of Intel's measurements. However, the FPGA-based solution achieved only 60% of the PCIe bandwidth due to the inefficiency of the memory controller, while the Asteralabs A1000 prototype reached an impressive 73.6% bandwidth efficiency, clearly outperforming Intel's FPGA-based solution.

4 Memory Capacity-bound Applications

One of the most significant advantages of integrating CXL memory into modern computing systems is the opportunity for significantly larger memory capacities. To elucidate the potential benefits, we focus on three particular use cases (1) key-value stores, a commonly used application in data centers. (2) Big data analytical application. (3) Elastic computing from cloud providers.



(a) Throughput of YCSB workload (b) Tail latency of YCSB-A (c) Tail latency of YCSB-C

Fig. 5. KeyDB YCSB latency and throughput under different configurations. (a) Average throughput of four YCSB workload under different system configuration. (b) Tail latency of YCSB-A (c) Tail latency CDF of YCSB-C, both reported by the YCSB client [34].

4.1 In-memory key-value stores

Redis [35] is an open-source in-memory key-value store and one of the most popular NoSQL databases. Redis employs a user-defined parameter, `maxmemory`, to limit its memory allocation for storing user data. Like traditional memory allocators (e.g., malloc()), Redis may not return memory to the system after key deletion, particularly if deleted keys were on a memory page with active ones. This necessitates memory provisioning based on peak demand, making memory capacity the major bottleneck for Redis deployments [36] in data centers. Google Cloud suggests keeping memory usage below 80% [37], whereas other sources recommend a limit of 75% [36].

Due to the substantial infrastructure costs for memory-only deployment, Redis Enterprise [38] is the commercial variant extensively supported by leading cloud platforms (e.g., AWS, Google Cloud, or Azure). It introduces "Auto Tiering"[39] to allow data overflow to SSDs, offering an economically viable option for database expansion beyond the limits of RAM capacity. Given that Redis Enterprise is not accessible on our experiment platform, we employ KeyDB as an alternative. KeyDB extends Redis's capabilities by adding KeyDB Flash, which uses RocksDB for persistent storage. The FLASH feature enables all data is written to the disk for persistence, with hot data remaining in memory as well as disk.

4.1.1 Methodology and Software Configurations. In our study, we investigate the performance effects of maximizing memory utilization on a KeyDB server. We deploy a single KeyDB instance on a CXL-enabled server configured with seven *server-threads*. Unlike Redis's single-threaded approach, KeyDB enhances performance by operating multiple threads to run the standard Redis event loop, akin to running several Redis instances simultaneously. We disable SNC and Transparent Hugepages and enable memory overcommitting within the kernel to minimize potential overhead from OS configurations. For KeyDB FLASH, we deactive all forms of compression in RocksDB to minimize software overhead.

Configuration	Description
`MMEM`	Entire working set in main memory.
`MMEM-SSD-0.2`	20% of the working set is spilled to SSD.
`MMEM-SSD-0.4`	40% of the working set is spilled to SSD.
`3:1`	Entire working set in memory (75% MMEM + 25% CXL, 3:1 interleaved).
`1:1`	Entire working set in memory (50% MMEM + 50% CXL, 1:1 interleaved).
`1:3`	Entire working set in memory (25% MMEM + 75% CXL, 1:3 interleaved).
`Hot-Promote`	Entire working set in memory (50% MMEM + 50% CXL), with hot page promotion kernel patches discussed in §2.

Table 1. Configurations used in capacity experiments.

Our empirical analysis uses the YCSB benchmark with four distinct workloads: (1) YCSB-A (50% read, 50% update) for update-intensive scenarios; (2) YCSB-B (95% read, 5% update) for read-heavy operations; (3) YCSB-C (100% read) for read-only tasks; and (4) YCSB-D (95% read, 5% insert) to simulate reading the most recent data. These workloads are tested under various system configurations as detailed in Table 1. Note that we use the term "MMEM" for main memory in order to separate it from CXL memory. For configurations utilizing SSD data spillover, we set the *maxmemory* parameter according to the portion of the workload expected to remain in memory. For Hot-Promote, we applied *numactl* to distribute half of the dataset across CXL memory while limiting the total main memory usage to half the dataset size. The experiments are conducted using a 1 KB key-value size, the YCSB default, with a Zipfian distribution for workloads A-C and the latest distribution for workload D. The total amount of working set data is 512 GB.

4.1.2 Analysis. Fig. 5 provides insights into the variations in throughput across different configurations. Notably, regardless of the specific workload, running the entire workload on MMEM consistently yields the highest throughput. This

outcome can be attributed to the nature of our workload, primarily constrained by memory capacity rather than memory bandwidth. The Hot-Promote configuration, which leverages the Zipfian distribution to identify frequently accessed keys as hot pages and migrates them from CXL to MMEM, performs nearly as well as running the workload entirely on MMEM. This demonstrates the effectiveness of the Hot-Promote approach in optimizing performance. In contrast, interleaving data access between CXL and MMEM leads to a noticeable performance decrease, resulting in a 1.2x to 1.5x slowdown compared to running the workload directly in MMEM. This performance drop is primarily due to the higher access latency, as evident in the tail latency plots for workload A and workload C (Fig. 5(b)(c)). MMEM-SSD-0.2 and MMEM-SSD-0.4 configurations perform the poorest, exhibiting nearly a 1.8x slowdown compared to the pure MMEM solution and a 1.55x slowdown compared to the CXL interleaving solution. This poor performance is mainly attributed to the high access latency required to retrieve data from the SSD. It's worth noting that our choice of a Zipfian distribution ensures that the working set is largely cached in MMEM. If the keys were distributed uniformly, we anticipate worse performance due to increased SSD access times.

4.1.3 Insights. Our study shows that the additional memory capacity provided by CXL can be a game-changer for applications like key-value stores constrained by traditional MMEM's capacity. Intelligent scheduling policies further accentuate the benefits, offering avenues for optimizing systems that leverage multiple memory types and simultaneously saving operation costs.

4.2 Spark SQL

Big Data plays a crucial role in the workloads managed by data centers. Due to the scale of data involved in Big Data analytical applications, memory capacity often becomes a bottleneck to the performance [40]. Take Spark [41], one of the common Big Data platforms, as an example: A typical query requires shuffling data from multiple tables for processing in the next stage. Operations like *reduceByKey()* first partition the data according to the key and then execute reduce operators on each key. Such shuffling operation involves disk I/O and network communication between multiple nodes, posing significant overhead on the query. In some cases, the performance of shuffling could dominate the performance of the workload [42]. During the shuffling process(Fig. 6), memory usage could grow beyond the capacity or certain threshold (e.g. `spark.shuffle.memoryFraction`). When this happens, Spark can be configured to spill data to disk to avoid the risk of out-of-memory failure. Since disk I/O is of magnitudes slower than memory, this could significantly impact the workload's performance.

4.2.1 Methodology and Software Configurations. In our experiment, we aim to test if we could reduce the number of servers needed for a specific workload with minimal effect on overall performance. Therefore, we compared the performance of Spark running TPC-H [43] on three servers without CXL memory expansion vs. on two servers but with CXL memory expansion. We assume the maximum amount of MMEM that could be used on each server is 512 GB, therefore with three servers, we have 1.5 TB MMEM and 1 TB CXL memory in total. In order to trigger data spill within the workload, we configured 150 Spark executors. Each Spark executor contains 1 core and 8 GB of memory. Therefore the total Spark application occupies 150 cores and 1.2 TB of memory. We generate a total of 7 TB TPC-H initial dataset. We continue to adhere to the configuration settings detailed in Table 1 as follows:

- MMEM only: We allocate 50 Spark executor and 400 GB on each of the **three** servers. In this case there is no data spilled to disk as each executor have sufficient amount of memory.
- MMEM/CXL interleaving: We distributed the same number of executors (150) across the **two** cxl servers, which has 1 TB (512 GB from each of the two CXL cards) plus 1 TB of MMEM (512 GB each). For example, in a configuration where MMEM and CXL memory usage is balanced (1:1 ratio), we allocated 75 Spark executors to use 600 GB MMEM while another 75 Spark executors to 600 GB CXL memory. In this case, there is also negligible amount of data spilled to the disk.
- Spill to SSD: To simulate conditions where executors would run out of memory and need to spill data to SSD

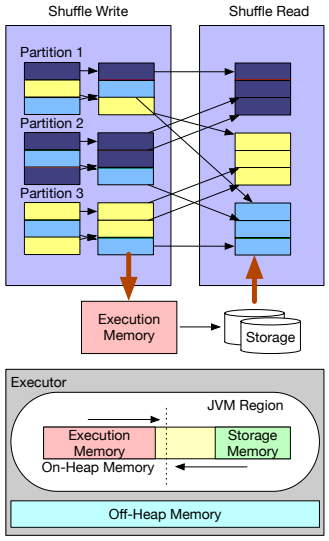
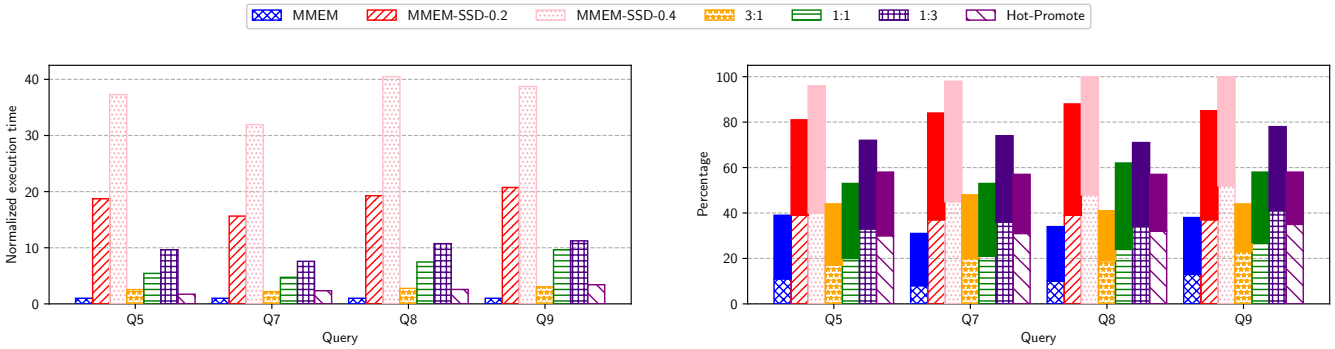


Fig. 6. Spark memory layout and shuffle spill. Each Spark executor possesses a fixed-size On-Heap memory, which is dynamically divided between execution and storage memory. If there is insufficient memory during shuffle operations, the Spark executor will spill the data to the disk.



(a) Normalized execution time of TPC-H queries

(b) Percentage of time spent on shuffle r/w

Fig. 7. Spark execution time and shuffle percentage. (a) Execution time of each TPC-H query normalized to the execution time running on MMEM. (b) The percentage of time spent of shuffle operation for each query. The solid bars represent shuffle writes, while hollow bars represent shuffle reads.

storage, we restrict the memory allocation of the Spark executors to either 80% or 60% of entire 1.2 TB MMEM. In this case, there will be around 320 GB and 500 GB data spilled to the disk respectively.

- Hot-Promote: same as prior experiment (§4.1).

We chose four specific queries (*Q*5, *Q*7, *Q*8, and *Q*9) from the TPC-H benchmark [43], recognized for their intensive data shuffling demands from prior studies [42], to evaluate our setup. Importantly, our measurements focused solely on the time to execute these queries, excluding any data preparation or server setup durations. We disabled SNC on all servers.

4.2.2 Analysis. Figure 7 illustrates variations in total execution time across different configurations. To provide a clear comparison, we normalized the total execution time against the best-case scenario, which involves running the entire workload in MMEM. Similar to the KeyDB experiments, the interleaving approach still exhibits a performance slowdown, ranging from 1.4x to 9.8x compared to the optimal MMEM-only scenario while using less number of servers. This performance degradation becomes worse as a larger proportion of memory is allocated to CXL. Nevertheless, it's crucial to note that even with this slowdown, the interleaving approach remains significantly faster than spilling data to SSDs. Figure 7(b) illustrates that shuffling overshadows the total execution time due to the intensification of data spill issues.

A notable difference between the KeyDB and Spark experiments is the performance of HotPromote. While it performs better in KeyDB, the Spark SQL experiment shows a more than 34% slowdown compared to MMEM. Unlike the Zipfian distribution in which the hottest keys are moved from CXL to DDR, there is a considerable amount of thrashing behavior within the kernel in the Spark SQL tests. We identify the root cause after thoroughly investigating the kernel patch implementation. In the initial version of the hot page selection patch [27], a sysctl knob "`kernel.numa_balancing_promote_rate_limit_MBps`" is used to control the maximum promoting/demoting throughput. Subsequent versions introduced an automatic threshold adjustment feature to this patch, aiming to strike a balance between the speed of promotion and migration costs. Nevertheless, this automatic adjustment mechanism appears to fall short in our Spark SQL evaluations. The TPC-H workload on Spark, which demonstrates reduced data locality, challenges the kernel's efficiency in promoting frequently accessed pages. This finding aligns with similar issues highlighted in prior research [11].

4.2.3 Insights. Our research indicates that utilizing CXL memory expansion offers a cost-efficient approach for datacenter applications. We postpone our detailed theoretical examination of the Abstract Cost Model to §6. Concurrently, although the hot-promote patch demonstrates significant advantages in key-value store workloads, its performance is notably lacking in Spark experiments. As system developers begin to enhance software support for CXL within the kernel, it is crucial to proceed with caution. System-wide policies can have varied impacts on applications, depending on their unique characteristics.

4.3 Spare Cores for Virtual Machine

One widely-used application within Infrastructure-as-a-Service (IAAS) is Elastic Computing [49]. Here, cloud service providers (CSPs) offer computational resources to users through virtual machines or container instances. Given the diverse needs of users, CSPs traditionally offer a variety of instance types, each characterized by different configurations of CPU cores, memory, disk, and network capacities. Generally, an "optimal" CPU-to-memory ratio, often cited as 1:4, is employed to balance computational and memory requirements (as per AWS guidelines [50, 51]). For example, an instance with 128 vCPUs would typically feature 512 GB of

Year	CPU	Max vCPU per server	Memory channels per socket	Max memory \TB	Required Memory (1 : 4) \TB
2021	IceLake-SP[44]	160	8xDDR4-3200	4	0.64
2022 (delayed)	Sapphire Rapids[45]	192	8xDDR5-4800	4	0.768
2023 (delayed)	Emerald Rapids[46]	256	8xDDR5-6400	4	1
2024+	Sierra Forest[47]	1152	12	4	4.5
2025+	Clearwater Forest[48]	1152	TBD	4	4.5

Table 2. Intel Processor Series.

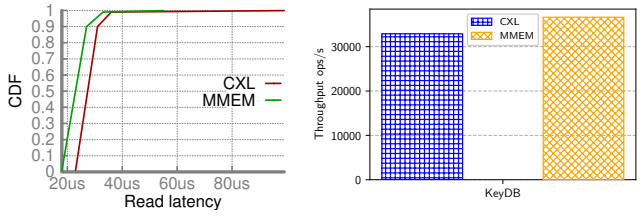
DDR memory. Advancements in server processor architecture and chiplet technology have spurred rapid increases in the number of cores available in a single processor package, driven in large part by the CSPs' aim to lower per-core costs. Consequently, 2-socket servers have seen their vCPU counts grow from 160 to 256 within the past two years (Table 2). This trend is projected to continue, reaching as many as 1152 vCPUs per server by 2025.

The surge in vCPUs exacerbates memory capacity bottlenecks, constrained by DDR slot limits, DRAM density, and the cost of high-density DIMMs. Intel's Sierra Forest Xeon, for example, supports 1152 vCPUs but is limited by motherboard design to less than 4 TB of memory, falling short of the typical 4.5 TB needed for VM provisioning [52]. This discrepancy makes maintaining a cost-effective vCPU-to-memory ratio challenging, resulting in underutilized vCPUs and lost revenue for CSPs. CXL memory expansion provides a solution by enabling memory capacity to scale beyond DDR limitations, ensuring optimal vCPU utilization and mitigating revenue losses for CSPs.

4.3.1 Methodology and Software Configurations. To assess the performance impact when an application operates exclusively on CXL memory, we replicate the KeyDB configuration from previous experiments (§4.1). We utilize *numactl* to allocate the KeyDB instance exclusively to MMEM or CXL memory. For our evaluation, the workload employed is YCSB-C, characterized by 1 KB key-value pairs and a total dataset size of 100 GB. SNC is disabled.

4.3.2 Analysis. The CDF of read latency (Fig. 8(a)) indicates that applications running on CXL experience a latency penalty of 9% – 27% which is less than the raw data fetching numbers in our previous measurements in §3. This is due to the processing latency within Redis. The throughput of running the entire workload on CXL memory is around 12.5% less compared to MMEM as show in Fig. 8(b).

Now consider a server operating at a sub-optimal vCPU-to-memory ratio of 1:3: (1) Due to inadequate memory, only 75% of the vCPUs can be sold at the optimal 1:4 ratio, resulting in a 25% revenue loss. Implementing CXL memory expansion enables the CSP to sell the remaining 25% of vCPUs at the optimal ratio. (2) Our benchmarks indicate that instances running on CXL memory perform 12.5% slower than those on DDR for common workloads such as Redis. Assuming a 20% price discount on such instances, CSPs could still recover



(a) CDF of KeyDB YCSB-C (b) Throughput of KeyDB YCSB-C

Fig. 8. KeyDB Performance with YCSB-C on CXL/MMEM.

approximately 80% of the lost revenue, equating to a 27% improvement in total revenue ($20/75 = 26.77\%$).

4.3.3 Insights. Given the sheer scale of Elastic Computing Service (ECS) applications in public clouds, the potential benefits of CXL memory expansion could be substantial. However, the challenge of maintaining an optimal virtual CPU (vCPU) to memory ratio, traditionally at 1:4, becomes more complex with the rapid increase in processor cores. This ratio, although standard, is under scrutiny for its applicability in future cloud computing paradigms. Notably, Bytedance's Volcano Engine Cloud [53] illustrates the variability in resource allocation by offering different ratios: 1:4 for general purposes, 1:2 for compute-intensive tasks, and 1:8 for memory and storage-intensive workloads. The impact of CXL memory expansion and pooling on these established ratios presents an intriguing avenue for exploration, raising questions about the adaptability of cloud providers to evolving hardware capabilities and the subsequent effect on resource allocation standards.

5 Memory Bandwidth-Bound applications

The other advantage of CXL memory expansion is its extra memory bandwidth. We use Large Language Model inference as an example to showcase how this can benefit real-world applications.

Recent work on LLM [54] shows that LLM inference is hungry for memory capacity and bandwidth. The limited capacity of GPU memory restricts the batch size of the LLM inference job and reduces computing efficiency since LLM models are memory-demanding. On the other hand, while CPU memory is high in capacity, it has lower bandwidth than GPU memory. The extra bandwidth and capacity offered by CXL memory make it a promising option for alleviating this bottleneck. For example, a CPU-based LLM inference

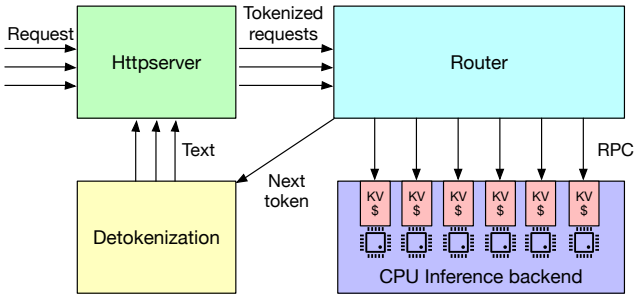


Fig. 9. LLM inference framework. The Httpserver receive requests and forward the tokenized requests to the CPU inference backend. The CPU inference backend serves the requests and reply the next token.

job can benefit from the extra bandwidth brought by CXL memory, and a CXL-enabled GPU device can also use the extra memory capacity from a disaggregated memory pool. Due to the lack of CXL support in current GPU devices, we experiment with LLM inference on CPU to study the implications of CXL memory's extra bandwidth. We also note that as LLM inference applications are agnostic to the underlying memory technologies, the findings and implications from our experiments are also applicable to the upcoming CXL 2.0/3.0 devices.

LLM Inference Framework. Mainstream Large Language Model (LLM) inference frameworks, such as vLLM [55] and LightLLM [56], do not support CPU inference. Recently, Intel introduced an LLM model named Q8chat [57], trained using their 4th Generation Intel Xeon® Scalable Processors. However, the inference code for Q8chat is not yet publicly available. To address this gap, we have developed our inference framework based on the open-source LightLLM framework [56] by replacing the backend with a CPU inference backend. Figure 9 illustrates our implementation. In our framework, the HTTPserver frontend receives LLM inference requests and forwards the tokenized requests to a router. The router is responsible for distributing these requests to different CPU backend instances. Each CPU backend instance is equipped with a Key-Value (KV) cache [58], a widely used technique in large language model inference. It's worth noting that KV caching, despite its name, differs from the traditional 'key-value store' in system architecture. KV caching occurs during multiple token generation steps, specifically within the decoder. During the decoding process, the model starts with a sequence of tokens, predicts the next token, appends it to the input, and repeats this generation process. This is how models like GPT [54] generate responses. The KV cache stores key and value projections used as intermediate data within this decoding process to avoid recomputation for each token generation. Prior research [58] has shown that KV caching is typically memory-bandwidth bound, as it is unique for each sequence in the batch, and different requests typically do not share the KV cache since the sequences are stored in separate contiguous memory spaces [59].

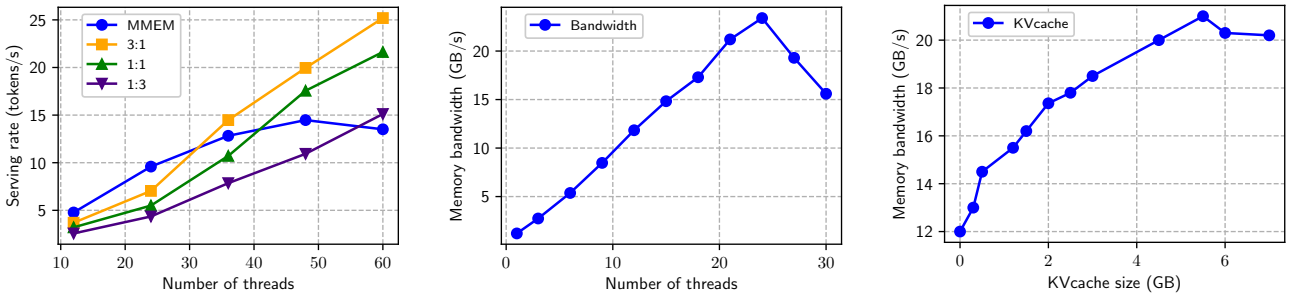
5.1 Methodology and Software Configurations

To investigate the benefits of CXL memory extension for applications with high memory bandwidth demands and limited MMEM bandwidth availability, we employ the SNC-4 configuration to divide a single CPU into four sub-NUMA nodes. Each node is equipped with two DDR5-4800 memory channels, facilitating an early memory bandwidth saturation of 67 GB/s (§3). We examine three distinct interleaving policies (3:1, 1:1, 1:3), detailed in Table 1. The CPU inference backend is configured with 12 CPU threads, and memory allocation is strictly bound to a single sub-NUMA domain. This domain includes two DDR5-4800 channels and a 256 GB A1000 CXL memory expansion module via PCIe. By binding allocations to a single node, we ensure the initial saturation of the DDR5 channels. Our experiments utilize the Alpaca 7B model [60], an advancement of the LLaMA 7B model, requiring 4.1GB of memory. The workload, derived from the LightLLM framework [56], includes a wide range of chat-oriented questions. A single-threaded client machine on a baseline server sends HTTP requests with various LLM queries to mimic real-world conditions. The client ensures continuous operation of the CPU inference backends by maintaining a constant stream of requests. The prompt context is set to 2048 bytes to guarantee a minimum inference response size. We progressively increase the CPU inference backend count to monitor the LLM inference serving rate (in tokens/s).

5.2 Analysis

Fig. 10(a) displays the inference serving rates across various memory configurations as the thread count, i.e., the number of CPU inference backends, increases. Initially, the serving rate improves almost linearly with available memory bandwidth. However, at 48 threads, MMEM bandwidth saturation limits the serving rate, whereas the interleaving configurations leverage additional CXL bandwidth for continued scaling. With a significant number of inference threads (60), an MMEM:CXL = 3:1 interleaving significantly surpasses the MMEM-only approach by 95%.

Interestingly, among the interleaving policies, configurations with a higher proportion of data in main memory demonstrate superior inference performance. Contrary to expectations, we observe that operating entirely on main memory is 14% less effective than a MMEM:CXL ratio of 1:3 beyond 64 threads. This outcome is notable given CXL's inherently higher latency and reduced memory bandwidth (§ 3). Fig. 10(b) charts the memory bandwidth utilization, as measured by the Intel Performance Counter Monitor (PCM) [32], with increasing CPU thread counts within a single CPU inference backend. Initially, bandwidth utilization grows linearly with thread count, plateauing at 24.2 GB/s for 24 threads. This trend allows us to estimate a bandwidth of approximately 63 GB/s at 60 threads, reaching 82% of the theoretical maximum. Our microbenchmark findings, as detailed in §3, indicate that this level of bandwidth utilization may lead to significant



(a) LLM inference serving rate vs. number of threads

(b) Memory bandwidth vs. number of threads for a single backend

(c) Memory bandwidth vs. KVcache size for a single backend

Fig. 10. CPU LLM inference.

latency spikes. These results corroborate the hypothesis that bandwidth contention plays a crucial role in the observed performance degradation.

Bandwidth contention may stem from either loading the LLM model or accessing the KV cache. Adjusting the prompt context to infinity enables the LLM model to continuously generate new tokens for storage in the KV cache. Fig. 10(c) illustrates the correlation between KV cache size and memory bandwidth consumption. The initial memory bandwidth of approximately 12 GB/s originates from I/O threads loading the model from memory. When storing information for a larger sequence of tokens in the KV cache, memory usage initially increases linearly. However, bandwidth utilization stops increasing beyond roughly 21 GB/s.

5.3 Insights

Interestingly, existing tiered memory management in the kernel does not consider memory bandwidth contention. Considering a workload that uses high main memory bandwidth(e.g., 70%), existing page migration policy(§2) tends to move data from slower tiered-memory (CXL) into MMEM, supposing that there is still enough memory capacity. As more data is written into the main memory, the memory bandwidth will continue to increase (e.g., 90%). In this case, the access latency will grow exponentially, resulting in an actual slowdown of the workload. This scenario will not be uncommon, especially for memory-bandwidth-bound applications (e.g., LLM inference). Therefore, the definition of tiered memory requires rethinking.

6 Cost Implications

Our comprehensive analysis in prior sections (§4, §5) reveals that the adoption of CXL memory expansion offers substantial benefits for data center applications, including comparable performance with operational cost savings. However, a significant hurdle in embracing such innovative technology as CXL lies in determining its Return on Investment (ROI). Despite having access to detailed technical specifications and benchmark performance results, accurately forecasting the Total Cost of Ownership (TCO) savings remains challenging. The complexity of simulating benchmarks at production scale, compounded by the limited availability of CXL hardware, exacerbates this issue. Traditional cost models in prior work [14], which could offer such forecasts, demand extensive internal and sensitive information that is often inaccessible. To overcome this barrier, we propose an Abstract Cost Model designed to estimate TCO savings independently of internal or sensitive data. This model leverages a select set of metrics obtainable through microbenchmarks, alongside a handful of empirical values that are simpler to approximate or access, providing a viable means to evaluate the economic viability of CXL technology implementation.

We use a capacity-bound application (Spark SQL) as an example to demonstrate how we develop our Abstract Cost Model, but our methodology can be extended to other types of workloads as well. For Spark SQL applications, the additional capacity enabled by CXL memory reduces the amount of data spilled to SSD and results in higher performance (throughput). This means fewer servers will be needed to meet the same performance target.

Given that the workload maintains a relatively consistent memory footprint (the size of the active dataset) during execution, we can approximate the execution time of the workload by dividing it into three distinct segments: (1) The segment processed using data stored in MMEM, (2) The segment processed using data stored in CXL memory, and (3) The segment processed using data that has been offloaded to SSD storage.

We first make these measurements from microbenchmarks on a single server:

- Baseline performance (P_s): Measure the throughput when (almost) all working set is spilled to SSD. The absolute number is not used in our cost model. Instead, we then normalize it to 1 in our cost model.
- Relative performance when the entire working set is in MMEM (R_d): Using the same workload, we measure the throughput when the entire working set is in MMEM and

Parameter	Description	Example Value
P_s	Throughput when (almost) entire working set is spilled to SSD on a server. Normalized to 1 in the cost model.	1
R_d	Relative throughput when the entire working set is in main memory on a server, normalized to P_s.	10
R_c	Relative throughput when the entire working set is in CXL memory on a server, Normalized to P_s.	8
D	The MMEM capacity allocated to each server. For completeness only, not used in cost model.	
C	The ratio of main memory to CXL capacity on a CXL server. E.g. 2 means the server has 2x MMEM capacity than CXL memory.	2
$N_{baseline}$	Number of servers in the baseline cluster.	
N_{cxl}	Number of servers in the cluster with CXL memory to deliver the same performance as the baseline.	
R_t	Relative TCO comparing a server equipped with CXL memory vs. baseline server. E.g. If a server with CXL memory costs 10% more than the baseline server, this parameter is 1.1.	1.1

Table 3. Parameters of our Abstract Cost Model .

normalize it to P_s to get the relative performance (i.e., how much faster compared to the baseline).

- Relative performance when the entire working set is in CXL memory (R_c): Using the same workload, we measure the throughput when the entire working set is in CXL memory, and normalize it to P_s to get the relative performance.

We then formulate our cost model using the parameters outlined in Table 3. For a working set size of W, the execution time of the baseline cluster could be approximated as the sum of two segments: 1) the segment that is executed with data in MMEM; 2) the segment that is executed with data spilled onto SSD.

$$T_{baseline} = \frac{N_{baseline}D}{R_d} + (W - N_{baseline}D)$$

The execution time of the cluster with CXL memory could be approximated in a similar way. It includes the segment that is executed with data in main memory, in CXL memory, and spilled to SSD respectively.

$$T_{cxl} = \frac{N_{cxl}D}{R_d} + \frac{N_{cxl}D}{CR_c} + (W - N_{cxl}D - \frac{N_{cxl}D}{C})$$

To meet the same performance target, $T_{baseline} = T_{cxl}$:

$$\frac{N_{baseline}D}{R_d} - N_{baseline}D = \frac{N_{cxl}D}{R_d} + \frac{N_{cxl}D}{CR_c} - N_{cxl}D - \frac{N_{cxl}D}{C}$$

With some simple transformations, we get the ratio between N_{cxl} and $N_{baseline}$:

$$\frac{N_{cxl}}{N_{baseline}} = \frac{CR_c(R_d - 1)}{R_cR_d(C + 1) - CR_c - R_d}$$

TCO saving can then be formulated as follows.

$$TCO_{saving} = 1 - \frac{TCO_{cxl}}{TCO_{baseline}} = 1 - \frac{N_{cxl}R_t}{N_{baseline}}$$

For example, suppose $R_d = 10$, $R_c = 8$, $C = 2$, we get $\frac{N_{cxl}}{N_{baseline}} = 67.29\%$ from the cost model. This means that by using CXL memory, we may reduce the number of servers by 32.71%. And if we further assume $R_t = 1.1$ (a server with CXL memory costs 10% more than the baseline server), the TCO saving is estimated to be 25.98%.

Our Abstract Cost Model provides an easy and accessible way to estimate the benefit from using CXL memory, providing important guidance to the design of the next-generation infrastructure.

Extending Cost Model for more realistic scenarios. In line with previous research [14], our Abstract Cost Model is designed to be adaptable, allowing for the inclusion of additional practical infrastructure expenses such as the cost of CXL memory controllers, CXL switches (applicable in CXL 2.0/3.0 versions), PCBs, cables, etc., as fixed constants. However, a notable constraint of our current model is its focus on only one type of application at a time. This becomes a challenge when a data center provider seeks to evaluate cost savings for multiple distinct applications, each with unique characteristics, especially in environments where resources are shared (for instance, through CXL memory pools). This scenario introduces complexity and presents an intriguing challenge, which we acknowledge as an area for future investigation.

7 Discussion

Our experiments concentrate on CXL 1.1 devices, yet the insights extend beyond this version. We also explore the relevance of our findings to future CXL architectures, including CXL 2.0 and 3.0, and discuss the anticipated evolution of CXL technologies.

7.1 CXL 2.0/3.0 and Beyond

Over the last two decades, we've seen the evolution from isolated computation and storage on single machines to the pooled resources that underpin today's cloud infrastructure, thanks to distributed computing and storage technologies. This shift to a disaggregated architecture has enabled computing and storage to scale independently, offering substantial cost savings for various large-scale workloads.

Looking ahead, we anticipate a similar transformation for memory resources. We foresee the next-generation data centers leveraging a disaggregated heterogeneous memory architecture with a unified address space, allowing workloads to dynamically allocate memory from a pooled resource and maintain a unified memory view across different memory types. This approach, decoupling memory scaling from other

GH200 memory tier	Resemblance to CXL
Local GPU HBM	Local DDR
Local CPU DDR	CXL memory expansion
Remote GPU HBM	CXL memory pooling
Remote CPU DDR	CXL memory pooling

Table 4. Comparison of memory architectures between Nvidia GH200 [61] and CXL memory .

resources like CPUs, promises enhanced elasticity and cost efficiency, with technologies such as CXL 2.0/3.0 playing a pivotal role due to their superior bandwidth, low latency, and scalable memory access features.

Furthermore, we predict this disaggregated memory architecture will merge with general computing and AI/ML workloads, reflecting recent architectural advancements in GPU and AI/ML accelerators, such as Nvidia's GH200 with its heterogeneous memory ([61]), Apple's M2 Ultra with its Unified Memory Architecture ([62]), and Google's TPUv4 with its globally addressable memory space ([63]). The Nvidia GH200, in particular, showcases a tiered memory structure akin to the CXL-based disaggregated memory architecture (4). We anticipate the disaggregated heterogeneous memory architecture will revolutionize not just general computing, but AI/ML workloads as well. However, realizing this vision faces several challenges:

- Hardware: Innovations in processors, such as those proposed in the CXL-centric architecture by Cho et al. ([6]), and the integration of emerging memory technologies like MRAM and ReRAM, are crucial. These technologies will bring new capabilities and challenges in management, scheduling, and placement algorithms.
- Software: The shift towards a disaggregated memory architecture necessitates significant changes in the software stack. This includes OS enhancements and the development of a unified memory management framework to handle allocation, placement, migration, provisioning, monitoring, and fault tolerance efficiently. This framework must support a variety of applications and hardware configurations with minimal performance overhead.
- Interconnect & Fabric Technology: The scalability of this architecture heavily relies on advancements in interconnect and fabric technologies. Current limitations of PCIe cables for CXL 1.1 memory expansion necessitate new solutions for scaling beyond single racks. Innovations like optical interconnects for PCIe [64] and Ultra Ethernet (UEC) [65] show promise for enabling large-scale, multi-rack, or even data center-wide disaggregated memory pools.

7.2 Other Datacenter Applications

Beyond the applications initially discussed, a wide array of data-center tasks stands to gain significantly from CXL memory expansion, particularly with the advancements in CXL 1.1 and 2.0 technologies. This includes Graph Neural Network (GNN) applications and genomics, where the immense memory requirements for processing entire graphs or extensive genomic sequences present substantial challenges. Enhancements in CXL's memory capacity and bandwidth can dramatically improve data access and processing speeds for these and similar data-intensive tasks. Moreover, the integration of Type-1 and Type-2 CXL devices opens up new avenues for optimizing interactions with heterogeneous accelerators, such as GPUs, which are pivotal in ML/AI workloads. This confluence of CXL memory expansion and accelerator technologies not only boosts data center scalability and efficiency but also elevates the performance of a broad spectrum of applications. From enhancing real-time analytics to enabling more efficient edge computing and IoT operations, CXL technology is instrumental in reducing latency, increasing throughput, and facilitating the real-time processing demands essential for advancing computational capabilities across various domains.

8 Related Work

The concept of memory disaggregation, highlighted in key studies [66, 67], aims to uncouple CPUs from local memory to optimize memory resource sharing in data centers, potentially alleviating memory bottlenecks. The integration of Compute Express Link (CXL) technology is explored for its potential to enhance system efficiency and reduce Total Cost of Ownership (TCO) [5, 6, 8, 9, 68]. Research on CXL spans various methodologies, including the use of NUMA servers as stand-ins for CXL memory [8], software simulators, and actual implementations on FPGA-based RISC-V CPUs [12]. Key industry players like Microsoft [8, 14] and Meta [9], alongside hardware vendors Intel [11] and Samsung [13], are moving towards adopting CXL, showcasing its performance advantages. Cho et al. [30] further propose a CXL-centric server processor architecture, suggesting the replacement of DDR controllers with CXL interfaces to capitalize on its higher bandwidth and efficiency.

9 Conclusion

We provide a comprehensive empirical evaluation of Compute Express Link (CXL) in real-world data center applications, filling a critical knowledge gap left by prior theoretical studies. Our findings reveal both the potential and limitations of CXL, offering actionable recommendations for its ongoing development to better serve data-centric computing environments. Based on our benchmarks, we also develop an Abstract Cost Model that can estimate the TCO savings without relying on internal or sensitive data, providing important guidance to the design of our next generation infrastructure.

Acknowledgement

We would like to thank our shepherd Kang Chen and anonymous EuroSys reviewers for their valuable comments and insightful feedback. We are also grateful to the dedicated members from ByteDance, both in the U.S. and China, for their significant contribution and tireless support in facilitating the setup of our experimental platform.

References

[1] Dez Blanchfield. The cloud native convergence: A new era of data-intensive applications. https://elnion.com/2023/06/05/the-cloud-native-convergence-a-new-era-of-data-intensive-applications/.

[2] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[4] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[5] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[6] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. A case for cxl-centric server processors, 2023.

[7] Leo Memory Connectivity Platform for CXL 1.1 and 2.0. https://www.asteralabs.com/wp-content/uploads/2022/08/Astera_Labs_Leo_Aurora_Product_FINAL.pdf.

[8] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa R. Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2022.

[9] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 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, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[10] 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. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[11] 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. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[13] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2):20–29, 2023.

[14] 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. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[15] Debendra Das Sharma. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *IEEE Micro*, 43(2):99–109, 2022.

[16] What Are PCIe 4.0 and 5.0? https://www.intel.com/content/www/us/en/gaming/resources/what-is-pcie-4-and-why-does-it-matter.html.

[17] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. An introduction to the compute express link (cxl) interconnect, 2023.

[18] Intel Corporation. Intel launches 4^{th} gen xeon scalable processors, max series cpus. https://www.intel.com/content/www/us/en/newsroom/news/.

[19] AMD Unveils Zen 4 CPU Roadmap: 96-Core 5nm Genoa in 2022, 128-Core Bergamo in 2023. https://wccftech.com/intel-clearwater-forest-e-core-xeon-cpus-up-to-288-cores-higher-ipc-more-cache/.

[20] Montage Technology. Cxl memory expander controller (mxc). https://www.montage-tech.com/MXC,accessedin2023.

[21] CZ120 memory expansion module. https://www.micron.com/products/memory/cxl-memory.

[22] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Intel Corporation. Intel Agilex® 7 FPGA and SoC FPGA I-Series. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7/i-series.html.

[24] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, page 21–30, New York, NY, USA, 2006. Association for Computing Machinery.

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

[26] NUMA balancing: optimize memory placement for memory tiering system. https://lore.kernel.org/linux-mm/20220221084529.1052339-1-ying.huang@intel.com/.

[27] Tiered memory: Hot page selection. https://lore.kernel.org/lkml/20220622083519.708236-2-ying.huang@intel.com/T/.

[28] Transparent Page Placement for Tiered-Memory. https://lore.kernel.org/all/cover.1637778851.git.hasanalmaruf@fb.com/.

[29] David L Mulnix. Intel® Xeon® Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[30] A. Cho and et al. A Case for CXL-Centric Server Processors. https://arxiv.org/abs/2305.05033.

[31] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT$\hat{2}$: Memory bandwidth regulation on hybrid NVM/DRAM platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, Santa Clara, CA, February 2022. USENIX Association.

[32] Intel Corporation. Intel® Performance Counter Monitor (Intel® PCM). https://github.com/intel/pcm.

[33] Intel Corporation. Intel Unveils Future-Generation Xeon with Robust Performance and Efficiency Architectures. https://www.intel.com/content/www/us/en/newsroom/news/intel-unveils-future-generation-xeon.html.

[34] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[35] Redis. . https://redis.io/.

[36] Tecton.ai. Managing your Redis Cluster. https://docs.tecton.ai/docs/0.5/setting-up-tecton/setting-up-other-components/managing-your-redis-cluster.

[37] Google Cloud. Memory management best practices. https://cloud.google.com/memorystore/docs/redis/memory-management-best-practices.

[38] Redis enterprise. https://redis.io/docs/about/redis-enterprise/, 2023.

[39] Auto Tiering Extend Redis Enterprise databases beyond DRAM limits. https://redis.com/redis-enterprise/technology/auto-tiering/#:~:text=Redis%20Enterprise's%20auto%20tiering%20lets,compared%20to%20only%20DRAM%20deployments.

[40] Xuechen Zhang, Ujjwal Khanal, Xinghui Zhao, and Stephen Ficklin. Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system. *J. Parallel Distrib. Comput.*, 120(C):369–382, oct 2018.

[41] Apache Spark. Unified engine for large-scale data analytics. https://spark.apache.org/.

[42] Chen Zou, Hui Zhang, Andrew A. Chien, and Yang Seok Ki. Psacs: Highly-parallel shuffle accelerator on computational storage. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 480–487, 2021.

[43] TPC-H is a Decision Support Benchmark. https://www.tpc.org/tpch/.

[44] Ice Lake SP: Overview and technical documentation. (n.d.). Intel. https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html.

[45] 4th Gen Intel Xeon Processor Scalable Family, sapphire rapids. (n.d.). Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html#gs.3m5uv2.

[46] McDowell, S. (2023, December 18). Intel launches 5th generation "Emerald Rapids" Xeon processors. Forbes. https://www.forbes.com/sites/stevemcdowell/2023/12/17/intel-launches-5th-generation-emerald-rapids-xeon-processors/.

[47] Kennedy, Patrick. "Intel Shows Granite Rapids and Sierra Forest Motherboards at OCP Summit 2023." ServeTheHome, 26 Oct. 2023,. www.servethehome.com/intel-shows-granite-rapids-and-sierra-forest-motherboards-at-ocp-summit-2023-qct-wistron.

[48] Mujtaba, H. (2023, December 1). Intel Clearwater Forest E-Core Only Xeon CPUs to offer up to 288 cores. https://wccftech.com/intel-clearwater-forest-e-core-xeon-cpus-up-to-288-cores-higher-ipc-more-cache/.

[49] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 236–243, 2010.

[50] Amazon EC2 M7a Instances. https://aws.amazon.com/ec2/instance-types/m7a/, 2023.

[51] Amazon EC2 M7i Instances. https://aws.amazon.com/ec2/instance-types/m7i/, 2023.

[52] Intel Shows Granite Rapids and Sierra Forest Motherboards at OCP Summit 2023. https://www.servethehome.com/intel-shows-granite-rapids-and-sierra-forest-motherboards-at-ocp-summit-2023-qct-wistron/.

[53] Elastic Compute Service, Volcano Engine, Bytedance. https://www.volcengine.com/product/ecs.

[54] G. Wong D. Patel. GPT-4 Architecture, Infrastructure, Training Dataset, Costs, Vision, MoE. https://www.semianalysis.com/p/gpt-4-architecture-infrastructure, 2023.

[55] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[56] Lightllm A Light and Fast Inference Service for LLM. https://github.com/ModelTC/lightllm.

[57] Julien Simon.Smaller is Better: Q8-Chat LLM is an Efficient Generative AI Experience on Intel® Xeon® Processors. https://www.intel.com/content/www/us/en/developer/articles/case-study/q8-chat-efficient-generative-ai-experience-xeon.html.

[58] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[59] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[60] Alpaca: A Strong, Replicable Instruction-Following Model. https://crfm.stanford.edu/2023/03/13/alpaca.html.

[61] Nvidia GH200 Datasheet. https://resources.nvidia.com/en-us-dgx-gh200/nvidia-grace-hopper-superchip-datasheet.

[62] Apple Introduces M2 Ultra. https://www.apple.com/newsroom/2023/06/apple-introduces-m2-ultra/.

[63] N. Jouppi and et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023.

[64] PCI-SIG explores an optical interconnect for higher PCIe performance. https://www.eenewseurope.com/en/pci-sig-explores-an-optical-connections-for-higher-pcie-performance/.

[65] Ultra Ethernet Consortium. https://ultraethernet.org/.

[66] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[67] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery.

[68] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. Performance evaluation on cxl-enabled hybrid memory pool. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–5, 2022.