Capacity-Latency Tradeoffs in CXL Memory Expander at Hyperscale

Haneul Park, *Student Member, IEEE*, Grant Ayers, Nam Sung Kim, *Fellow, IEEE*, Philip Levis, *Member, IEEE*, and Brian Morris

***Abstract*—DRAM dominates the cost of hyperscale datacenter servers today and AI-driven memory supply shortages are pushing costs even higher. CXL memory expanders are a new technology to reduce server costs by reusing legacy DDR DRAM modules from older servers. Memory expanders further save cost through inline compression that increases the effective capacity. This paper investigates two critical design points of a memory expander: the compression algorithm and the compression block size. Both design points introduce tradeoffs between latency and capacity that involve complex interactions with the memory expander architecture, access patterns to the device, and the entropy of stored data. Current commercial devices use simple, fast compression and a 4 kB block as the default. Using a compressibility benchmark and the latest traces of cold memory accesses from a hyperscaler, we find that using a different compression algorithm and block size can simultaneously increase expander capacity *and* reduce access latency. The key insight is counter-intuitive: using a slower, more complex compression algorithm improves latency because the reduction in DRAM traffic more than offsets the additional decompression time.**

***Index Terms*—CXL, far memory, datacenter, compression.**

I. Hyperscale CXL Tiered Memory Expander

RISING memory costs and increasing capacity demand from memory-intensive workloads have become major challenges in hyperscale datacenters. DRAM now dominates the total cost of ownership (TCO) of datacenter servers [1], [2]. This trend has been exacerbated by a sharp increase in server DRAM prices, caused by manufacturing capacity shift toward HBM for AI datacenters [3], [4]. One solution for datacenters to reduce memory cost is to reuse older, slower legacy DRAM modules from retired servers by attaching them to a CXL-based memory expander, such as the Open Compute Project (OCP) device by Google and Meta [5].

A defining feature of the OCP memory *expander* is hardware compression to increase effective memory capacity, which distinguishes it from CXL-based memory pooling systems. Fig. 1 illustrates the architecture of the expander. When the CPU moves a page to the expander, the device breaks and compresses it at block granularity before storing it in DRAM. When a CPU requests a cache line, the expander fetches and decompresses the associated block from DRAM into an SRAM cache and supplies the cache line to the CPU. Because of additional latencies from CXL interface and decompression, datacenter servers use expanders to store cold pages [6] that are accessed infrequently.

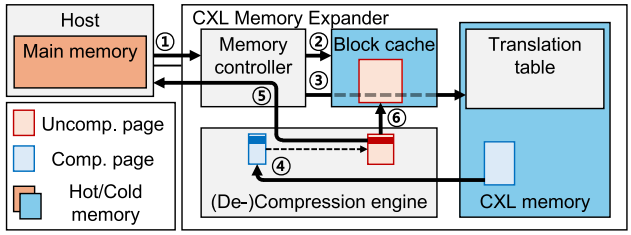

Fig. 1. Architectural diagram of a CXL memory expander. Worst-case read-to-use path (a block cache miss) is shown.

Expanders have two primary mechanisms to trade off between memory capacity and access latency: compression algorithm and block size. More complex algorithms offer higher compression at the cost of increased decompression latency. The compression block size is the unit that the expander compresses and decompresses. Larger blocks improve compression, but take longer to decompress. However, these tradeoffs are further complicated by *read amplification:* servicing a request for a single cache line requires reading an entire block from DRAM. Larger blocks and simpler algorithms incur higher read amplification, which may saturate memory bandwidth and, in turn, increase access latency. If most of the cache lines in a block are accessed before the decompressed block is evicted from the SRAM cache, however, the read is amortized across many accesses. The interplay of these tradeoffs depends greatly on the access patterns to the cold memory and underlying data compressibility.

Current commercial expander implementations [7] use the extremely simple LZ4 compression algorithm and a 4 kB block size. These design decisions are inherited from well-known approaches in software-based memory compression, such as zswap [8] or disaggregated memory [9]. Expanders, however, differ from software systems in three ways: they do not require software-handled page faults, they implement compression algorithms in hardware, and offer much lower latencies. Whether expanders should use the same block sizes and algorithms as software systems is an open question.

This work proposes the first evaluation of how compression algorithms and block sizes affect the capacity and latency of expanders in hyperscale servers. Section II examines the compressibility of cold pages at sub-page granularity. Section III explores the locality of cold page accesses in hyperscale deployments. The key finding is that more complex

Received 29 January 2026; revised 16 March 2026; accepted 19 March 2026. Date of publication 23 March 2026; date of current version 6 April 2026. This work was supported in part by PRISM and CogniSense, in part by two of the seven centers in JUMP 2.0, and in part by Semiconductor Research Corporation (SRC) program sponsored by DARPA. *(Corresponding author: Haneul Park.)*

Haneul Park and Nam Sung Kim are with the University of Illinois Urbana-Champaign, Urbana, IL 61801 USA (e-mail: hnpark2@illinois.edu; nskim@illinois.edu).

Grant Ayers and Brian Morris are with the Google LLC, Mountain View, CA 94043 USA (e-mail: granta@google.com; bsmorris@google.com).

Philip Levis is with the Stanford University, Stanford, CA 94305 USA, and also with the Google LLC, Mountain View, CA 94043 USA (e-mail: plevis@google.com).

Digital Object Identifier 10.1109/LCA.2026.3676988

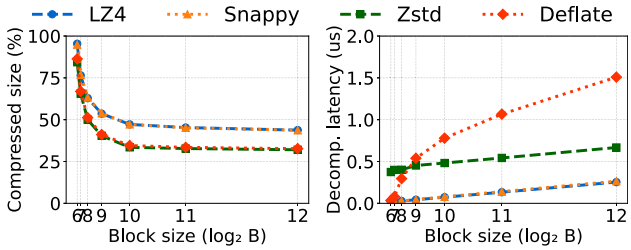

Fig. 2. Average compressed size and decompression latency of the HyperCompressBench corpus.

compression (*i.e.*, zstd) simultaneously increases capacity by 30% *and* reduces latency by 12%. This challenges the conventional view that simple algorithms are preferable as decompression lies on the critical path. Instead, memory bandwidth and queueing can dominate end-to-end latency in memory expanders. Therefore, the reduced read amplification of more complex compression outweighs its longer execution time. These results indicate that future expander specifications should be designed differently.

II. Algorithm and Block Size Tradeoffs

A. Methodology

We use HyperCompressBench [10] to generate representative expander data, reflecting compressibility observed in hyperscaler fleets. We measure the compression size and latency for seven block sizes (64B–4kiB, in powers of 2) and four compression algorithms (LZ4 [11], Zstd [12], Snappy [13], and Deflate [14]). LZ4 and Snappy are optimized for low latency, while Deflate and Zstd target higher compression ratios. We measure decompression latency using FPGA-based designs [15] and derive a scaling factor by comparing them against an ASIC-based Deflate decompressor [16], which also aligns with frequency scaling.

We evaluate both the raw performance of compression algorithms and their performance in an expander, including the following details. *Incompressible blocks* are identified upon compression; If a block's compressed size is >80% of its original size, the expander stores it uncompressed. Incompressible blocks take longer to read from DRAM but bypass decompression. Furthermore, expanders manage memory in fixed-sized *chunks*, such that compressed blocks are quantized into chunks, lowering the effective capacity.

B. Raw Performance Analysis

Fig. 2 shows that Zstd and Deflate reduce compression size by 13–37% over LZ4 and Snappy. Algorithms with the same computational foundations yield similar compressed sizes. All algorithms compress larger blocks more, but there are diminishing returns after 1KiB, and the space savings of tiny blocks are small. Heavyweight algorithms have higher latencies. Zstd's minimum latency, 370 ns, is more than an order of magnitude higher (21×) than the others. Deflate's latency for tiny blocks is small and competitive with LZ4 and Snappy. For larger blocks,

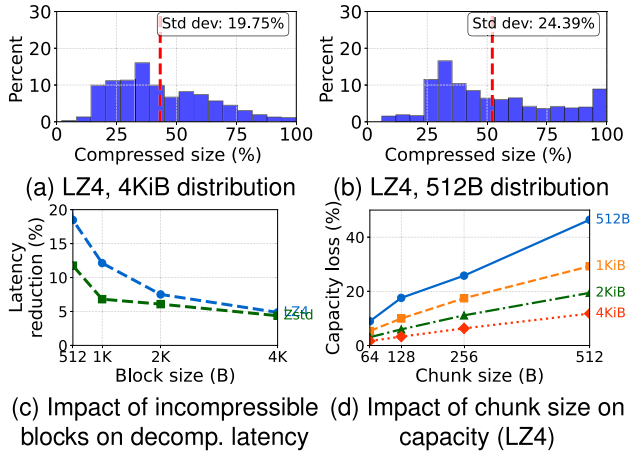

Fig. 3. Compressed-size distributions and their impact on latency and capacity.

Deflate and Zstd have $> 2.5\times$ higher latency than LZ4 and Snappy.

C. Performance in the Memory Expander

The distribution of compressed size and incompressibility also affects latency and capacity. Fig. 3(a)–3(b) shows the distribution of sizes of compressed blocks for LZ4 with 4KiB and 512B block sizes on an expander. The incompressibility rate increases as block size decreases: for 512B, 18.5% of blocks are incompressible, but 4.8% of 4KiB blocks are incompressible. The greater incompressibility rate translates into a decompression latency reduction of 18% (Fig. 3(c)).

Fig. 3(d) shows how chunk quantization impacts effective capacity of LZ4 compression. Using smaller block sizes wastes more space, demanding finer-grained memory management. The distribution of compressed sizes affects how much capacity quantization wastes. For instance, Fig. 3(b) shows the distribution unfavorable to chunk quantization, where 27% of the blocks compress to just over 128 bytes (25% compressed size). If the chunk size is 64B, these blocks have to be stored as 192B, while a 128B chunk size would require 256B.

III. Quantifying Tradeoffs at Hyperscale

Section II shows how algorithms and block sizes affect the required space in expander DRAM, and raw decompression latency. While the actual performance of an expander depends on these results, it is more complicated. Three architectural properties explain why large decompression latency does not necessarily translate into a proportional increase in total access latency. First, the CPU can consume a cache line immediately after decompression; traces that access earlier cache lines within a block achieve higher performance than those accessing later ones. Second, once a block is decompressed into the SRAM cache, the cache handles subsequent accesses, reducing both latency and DRAM bandwidth use. Third, memory fetches themselves incur substantial latency: algorithms with higher compression ratios issue fewer DRAM reads, thereby reducing both read latency and DRAM bandwidth pressure. Therefore, a

TABLE I
PERCENTAGE DIFFERENCE OF IDLE LATENCY BETWEEN TARGET AND SIMULATED LATENCIES

	Cache hit	Incompressible block	Compressed block
Diff. (%)	0.00%	+2.61%	-3.59%

detailed understanding of the capacity–latency tradeoff requires accounting for access patterns.

Expander memory access patterns differ significantly from the patterns that individual programs generate. Datacenter servers run many independent workloads concurrently to achieve high utilization [17], and the OS makes cold-page decisions globally. Furthermore, the access patterns of cold pages are a skewed version of average behavior across all pages [8]. This section evaluates the end-to-end performance of different memory expander configurations using cycle-accurate simulation and traces from the Google fleet.

A. Methodology

We use memory traces from an Intel Optane persistent memory-based far memory (equivalent to Google's [18]) deployed in a hyperscaler fleet. We sampled 1% of memory accesses using Linux `perf`, as production-server overhead constrained the sampling rate. Each trace records the physical addresses and timestamps of accesses to cold pages: cache and host DRAM accesses are excluded. There are ten total traces, Trace-0 through Trace-9; each trace is from a different server running a different mix of hundreds of concurrent workloads.

We augment each trace with a compressed size and decompression latency pair sampled from the results in Section II. The performance and privacy concerns preclude us from measuring compressibility in production servers. However, biased assignments that favor higher or lower compressibility for high-miss pages changed average latency by only 3–5%, leaving the qualitative conclusion unchanged. As Section II suggests, Snappy and LZ4 can be used interchangeably, and Zstd is faster than Deflate with block sizes that offer sufficient compression, and for brevity, we examine only LZ4 and Zstd. The simulator is based on the *gem5* framework, ensuring that the memory configuration and idle latency closely match the actual device [7] as shown in Table I. The main memory consists of four 8 GB DDR4-2600 DIMMs, with one DIMM per channel. The block cache is a 16 MB, 8-way set-associative LRU cache with an access latency of 30ns [5]. Each compression configuration has sufficient decompression engines to match the full DRAM bandwidth [5].

B. Optimal Compression Configuration

Table II shows the average access latency reduction and effective capacity increase of compression configurations across all of the traces compared to the baseline configuration of LZ4 with 4KiB blocks. LZ4 with a 512 B block size achieves the lowest average latency (33.8% reduction), at the cost of a 24.4% reduction in effective capacity relative to the baseline.

TABLE II
AVERAGE ACCESS LATENCY REDUCTION AND CAPACITY INCREASE OF ALL TRACES. BOLDFACED ENTRIES ARE PARETO-OPTIMAL CONFIGURATIONS, AND UNDERLINED VALUES ARE THE MOST IMPROVED METRICS IN EACH ALGORITHM OVER THE BASELINE (LZ4/4KIB).

	LZ4		Zstd	
Block size	Latency reduction	Capacity increase	Latency reduction	Capacity increase
4KiB	0%	0%	**11.5%**	**35.8%**
2KiB	**16.1%**	**-4.73%**	**12.1%**	**30.0%**
1KiB	**27.3%**	**-10.7%**	5.26%	21.8%
512B	**33.8%**	**-24.4%**	-31.0%	-2.27%
256B	33.4%	-36.8%	-51.3%	-28.0%
128B	32.5%	-53.4%	-82.3%	-47.8%
64B	30.5%	-55.5%	-55.6%	-55.5%

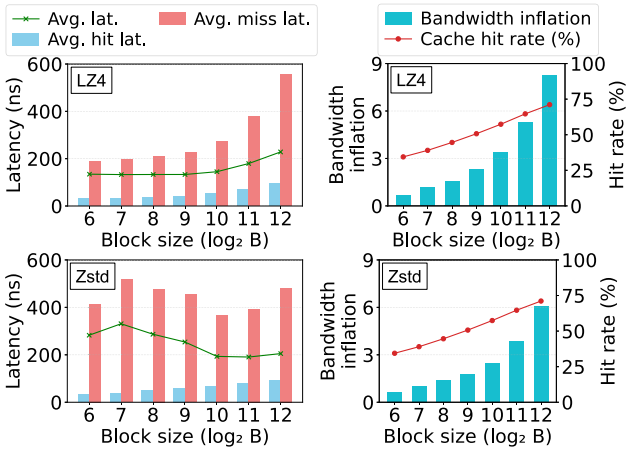

Fig. 4. Latency breakdown and bandwith inflation of LZ4 and Zstd on a representative trace (Trace-3).

Zstd with a 4KiB block size increases capacity by 35.8% over the baseline. Surprisingly, it also *decreases* latency by 11.5%. This improvement is despite Zstd's much higher single-block decompression latency compared to LZ4 (1 μs vs 250ns). Zstd's latency improves slightly with 2KiB blocks, to 12.1% lower than the baseline. The rest of this section explores these tradeoffs and the counterintuitive result that using large blocks, Zstd can simultaneously increase capacity and reduce latency.

C. Bandwidth Inflation, Queuing, and Delayed Hits

To explain how Zstd can simultaneously improve both capacity and latency, Fig. 4 shows a more detailed decomposition of latency for LZ4 and Zstd on a representative trace, Trace-3, for different block sizes. To better reason about the behavior of these algorithms, we define a *bandwidth inflation* metric as a ratio, how much the expander reads from DRAM to load a block, divided by how many bytes of the block the host requests. Read amplification considers the amount of data read from DRAM to serve a single cache line, while bandwidth inflation considers the fact that a single decompressed block can serve multiple cache hits. In theory, if every cache line from a compressed block were accessed, it would be possible to have a bandwidth inflation below 1.

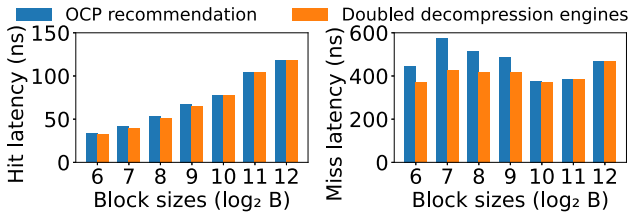

Fig. 5. Average hit and miss latency with the recommended and doubled number of decompression engines for Zstd.

Fig. 4 shows that Zstd has a much smaller bandwidth inflation than LZ4. Increasing the block size increases cache hit rates and bandwidth inflation for both algorithms. For example, on average, LZ4 compresses a 4KiB block to 44.5% of its original size, with read amplification of 28× (44.5% × 64). With an average block cache hit rate of 71%, 3.4 cache lines are served from a 4KiB block, resulting in bandwidth inflation of $64 \times 44.5\% \times 29\%$=8.2. Due to its better compression, Zstd with the same block size has a bandwidth inflation of 6.0.

Fig. 4 shows that average hit latency increases with block size for both algorithms, despite the fact that SRAM cache read performance is independent of block size. This is due to the phenomenon of delayed hits [19]. If a request arrives for a block that is already being decompressed, it counts as a hit. The latency to serve the hit, however, includes the remaining time to decompress the block. Smaller blocks see fewer delayed hits because it is less likely that a future access will land within the block. Average miss latency increases with larger blocks in LZ4, as higher bandwidth inflation increases queueing delay on DRAM accesses. For LZ4, average latency grows more slowly than hit or miss latency as larger blocks increase the cache hit rate.

Zstd's complex latency behavior is due to how its high minimum decompression latency (Fig. 2) interacts with the limited number of decompression engines. For blocks larger than 512B, miss latency grows primarily due to memory queueing delays. For blocks smaller than 1KiB, however, the miss latency increases. This occurs because, as the block size decreases, the expander has to decompress more blocks. For instance, when the block size halves from 1KiB, the number of blocks to decompress rises by 16% (increase in cache miss rate), but per block decompression time decreases by only 4%. Decompression becomes compute-bound rather than memory-bound, increasing queueing on decompression engines. As blocks become very small (from 128B to 64B), however, all blocks are incompressible, and this reduces load on the decompression engines.

The fact that the decompression latency of smaller blocks is compute-bound suggests that the OCP guideline of matching decompression throughput with memory bandwidth is insufficient. Fig. 5 shows how latency changes for Zstd with twice as many decompression engines. Queueing on decompression engines is reduced, and the smaller blocks see much smaller increases in miss latency. Average hit latency is also reduced, as delayed hit latency decreases.

D. End-to-End Analysis

Zstd simultaneously reduces access latency and increases the effective capacity of an expander. Because LZ4 compresses blocks less, it must read more from DRAM to serve the same cache line. This greater bandwidth use introduces significant queueing delays as the DRAM bandwidth saturates.

This analysis shows that improved compression provides an additional benefit beyond greater effective capacity: it also uses DRAM bandwidth more efficiently. LZ4 is required by the OCP specification for its low latency, but in real systems that have to deal with queuing and bandwidth limitations, and under real access workloads, DRAM bandwidth efficiency is more important than computational efficiency.

REFERENCES

[1] S.-H. Lee, "Technology scaling challenges and opportunities of memory devices," in *Proc. 2016 IEEE Int. Electron Devices Meeting*, 2016, pp. 1.1.1–1.1.8.
[2] D. S. Berger et al., "Design tradeoffs in CXL-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, Mar./Apr. 2023.
[3] Reuters, "The AI frenzy is driving a memory chip supply crisis," 2025. [Online]. Available: https://www.reuters.com/world/china/ai-frenzy-is-driving-new-global-supply-chain-crisis-2025-12-03/
[4] N. World, "Samsung warns of memory shortages driving industry-wide price surge in 2026," 2026. [Online]. Available: https://www.networkworld.com/article/4113772/samsung-warns-of-memory-shortages-driving-industry-wide-price-surge-in-2026.html
[5] P. Chauhan, C. Petersen, B. Morris, and J. Glisse, "OCP hyperscale cxl tiered memory expander specification," Open Compute Project Foundation, 2023. [Online]. Available: https://www.opencompute.org/documents/hyperscale-tiered-memory-expander-specification-for-compute-express-link-cxl-1-pdf
[6] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 631–644, Apr. 2017.
[7] Marvell Technology, "Marvell structera a 2504 CXL near-memory accelerator," 2024. Accessed: Jul. 11, 2025. [Online]. Available: https://www.marvell.com/products/cxl.html
[8] A. Lagar-Cavilla et al., "Software-defined far memory in warehouse-scale computers," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 317–330.
[9] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Trans. Comput.*, vol. 69, no. 6, pp. 844–855, Jun. 2020.
[10] S. Karandikar et al., "CDPU: Co-designing compression and decompression processing units for hyperscale systems," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, Art. no. 39.
[11] Y. Collet, "Lz4 - Extremely fast compression algorithm," 2024. Accessed: Jul. 11, 2025. [Online]. Available: https://github.com/lz4/lz4
[12] Facebook, "Zstandard - Real-time data compression algorithm," 2024. Accessed: Jul. 11, 2025. [Online]. Available: https://facebook.github.io/zstd/
[13] Google, "Snappy: A fast compressor/decompressor," 2024. Accessed: Jul. 11, 2025. [Online]. Available: https://github.com/google/snappy
[14] J.-L. Gailly and M. Adler, "zlib home site - A massively spiffy yet delicately unobtrusive compression library," 2024. Accessed: Jul. 11, 2025. [Online]. Available: https://www.zlib.net/
[15] Xilinx, "Vitis libraries," 2024. Accessed: Jul. 11, 2025. [Online]. Available: https://github.com/Xilinx/Vitis_Libraries
[16] J. Kang, Q. Xia, I. Jeong, Y. Park, and N. S. Kim, "Intel in-memory analytics accelerator: Performance characterization and guidelines," in *Proc. 2025 IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2025, pp. 1–13.
[17] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 18.
[18] P. Duraisamy et al., "Towards an adaptable systems architecture for memory tiering at warehouse-scale," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 727–741.
[19] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 495–513.