Rethinking Compression for CXL Memory Expanders at Hyperscale

Haneul Park*
University of Illinois
Urbana, USA
hnpark2@illinois.edu

Grant Ayers
Google
Sunnyvale, USA
granta@google.com

Nam Sung Kim
University of Illinois
Urbana, USA
nskim@illinois.edu

Philip Levis
Google
Sunnyvale, USA
plevis@google.com

Brian Morris
Google
Sunnyvale, USA
bsmorris@google.com

***Abstract*—DRAM dominates datacenter server cost, and recent AI-driven memory supply shortages are pushing costs higher. OCP CXL memory expanders address this challenge by attaching legacy DRAM as a slower capacity tier and increasing effective capacity through hardware compression.**

This paper explores two critical memory expander design points: the compression algorithm and the block size. Using cold memory traces measured in a hyperscaler fleet, we find that a more complex algorithm (Zstd) increases effective capacity by 30% while simultaneously reducing access latency by 12% compared to LZ4. This overturns the conventional assumption that memory compression should favor simple algorithms to minimize decompression latency. More complex compression reduces DRAM traffic, which more than offsets the higher decompression time.

We find that benchmarks widely used in tiered memory research exhibit fundamentally different locality from production cold memory, leading to potentially misleading design conclusions. To bridge this gap, we propose *COLA*, a parameterized model of COld memory LocALity with an integrated trace generator that enables systematic exploration of diverse access patterns. The performance improvements hold across a wide range of access traces, suggesting that next-generation expanders should adopt different algorithms and block sizes. *COLA* is open-sourced at https://github.com/google-systems-research/cold-memory-model.

***Index Terms*—Tiered memory, CXL, Compression.**

I. Introduction

Today, DRAM accounts for up to 50% of the total cost of ownership in datacenter servers [1], [2]. AI-driven supply shortages are pushing DRAM costs even higher [3], [4]. To reduce cost, hyperscalers are exploring memory tiering, which places infrequently accessed cold pages in a slower, cheaper tier. The Open Compute Project (OCP) Hyperscale CXL Tiered Memory Expander, proposed by Google and Meta [5], for example, reuses legacy DRAM from retired servers by attaching them to a CXL memory expander card.

A memory expander increases its effective capacity by compressing cold pages before storing them. This capacity increase has two performance costs. Decompression adds latency to memory accesses. Compression also introduces *read amplification*: the memory expander compresses data at block granularity (*e.g.*, 4KiB), so to serve a single 64B host request, the memory expander must fetch and decompress an entire block. To mitigate the costs, expanders cache decompressed blocks in an on-device SRAM buffer (block cache).

The choice of compression algorithm and block size affects the performance costs and capacity gain. Larger blocks expose more data redundancy, and more complex algorithms exploit redundancy more effectively, both yielding better compression and further increasing the capacity. However, their effect on latency pulls in opposing directions. Larger blocks increase both decompression and memory fetch latency. More complex algorithms also increase decompression latency but compress data more, thereby reducing memory reads and fetch latency. The complex capacity-latency tradeoffs make finding the right compression configuration non-trivial.

The optimal configuration also varies depending on how the host accesses the memory expander. This is because access patterns determine how effectively the block cache absorbs decompression and read amplification costs. If many cache lines within a block are accessed before eviction, larger blocks increase cache hit rates. On the other hand, if cache misses arrive in bursts, read amplification creates a large instantaneous bandwidth demand, and smaller blocks reduce the bandwidth needed.

These tradeoffs can affect performance greatly, but commercial memory expanders [6], [7] and recent work [8] deploy only the simplest compression algorithms – a choice inherited from software systems [9], [10]. In software systems, decompression is implemented in software and triggered by page faults, so the simplest algorithm is the only viable option given the already large latency. Memory expanders, however, operate at an order-of-magnitude lower latency: they decompress in hardware, serve data through a direct load/store interface without page faults, and cache decompressed blocks on-device.

In this paper, we evaluate compression configurations for OCP CXL memory expanders using cold memory traces from a production hyperscaler fleet. We focus on configurations that the proposed architecture already supports or can adopt with minimal hardware changes. Fleet traces ground our evaluation in real production behavior. Yet the diversity of a fleet spanning millions of servers and services means that a rigorous evaluation must go beyond any fixed set of traces.

We find that benchmarks widely used in tiered memory research do not fill this gap. Cold memory in datacenters

*This work was performed while Haneul Park was at Google. The remaining authors are listed in alphabetical order.

© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

arises from the aggregation across hundreds of co-running workloads, producing locality characteristics that differ fundamentally from any single benchmark. Benchmarks also behave more like load tests, with far higher access intensity than production cold memory [11]–[13] and runtimes of minutes rather than the hours needed for cold memory patterns to emerge [14].

To address the limitations of benchmarks, we propose *COLA*, a novel parameterized model of **CO**ld memory **L**oc**A**lity with an integrated trace generator. Using fleet traces and *COLA*, this paper presents the first evaluation of compression configurations in OCP memory expanders at hyperscale. This paper makes three contributions:

Compression evaluation at hyperscale. Using cold memory traces from a large hyperscaler, we find that a more complex algorithm (*i.e.*, Zstd) with a 2KiB block simultaneously increases capacity by 30% and reduces access latencies by 12% over LZ4 with a 4KiB block. Even for 4KiB blocks, Zstd has a lower latency than LZ4, because its better compression reduces the amount of data read from the memory. For memory expanders, memory bandwidth can be the bottleneck; improving memory bandwidth efficiency is more important than decompression speed.

Benchmark representativeness. These tradeoff results do not transfer to benchmarks typically used to evaluate tiered memory systems. Benchmarks generate orders of magnitude more accesses and exhibit fundamentally different locality from fleet traces, Zstd showing less or no latency benefits. Benchmarks, therefore, cannot predict real system behavior and mislead design conclusions.

Modeling cold memory accesses. We propose *COLA* for fast and accurate design space exploration of datacenter tiered memory systems. *COLA* captures locality profiles that strongly affect tiered memory performance but are missing in prior models [15]–[17]: temporal clustering of cache misses, per-page access counts, and access sparsity. Locality profiles are encoded into a few parameters. This controllable parameterization enables a systematic exploration of diverse cold memory access patterns. Using *COLA*, we show that the benefits of a more complex compression generalize broadly: for 75% of generated traces, Zstd outperforms the industry-default configuration on both capacity and latency.

II. Hyperscale Tiered Memory Expander

This section describes the OCP Hyperscale Tiered Memory Expander and explains two critical design points: compression algorithm and block size.

A. Hardware Compression and Access Latency

A defining feature of the memory expander—driven by its goal of increasing effective capacity—is hardware compression, which distinguishes it from CXL-based memory pools [18], [19]. The CXL interface enables the host to access the compressed memory transparently through the load/store interface.

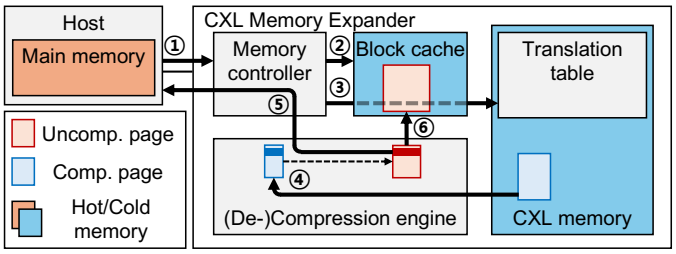


Fig. 1: Architectural diagram of a CXL memory expander. Black arrows show the worst-case read-to-use path.

Because CXL and (de-)compression add latency to memory accesses, datacenters use memory expanders to store only cold pages. Cold pages are pages that the operating system observes to be inactive for some time or predicts to be accessed infrequently or with low reuse [20]. Memory expander performance depends on the compressibility of cold pages and on how it is accessed. This makes the design tradeoffs of memory expanders fundamentally different from general-purpose memory compression such as MXT [21].

The key performance consideration for a memory expander is read latency. While memory writes can occur asynchronously, memory reads are on the critical path and must complete as soon as possible. Figure 1 illustrates the worst-case read request to the memory expander, which involves delays in the main components of the device. First, the host CPU accesses a cache line stored in the expander (①). This cache line is not in the SRAM block cache (②), so the expander translates the host physical address to the local address of the corresponding compressed block (③). The translated local address is forwarded to the decompression engine, which fetches and decompresses the corresponding block (④). The cache line is returned to the host immediately after its portion is decompressed (⑤). The expander places the decompressed block into the block cache (⑥).

B. Capacity-Latency Tradeoffs in Memory Expander

Both the effective capacity and the latency depend on how the memory expander compresses data. We explore the two key dimensions of these tradeoffs: block size and algorithm.

Block size tradeoffs. The block size is the unit that the memory expander compresses and decompresses. For example, with a block size of 1KiB, each 4KiB page is divided into four 1KiB blocks, and each block is compressed individually before being stored in the expander. Larger blocks expose more data redundancy and are compressed more, increasing effective capacity. The latency side is more complex.

Larger blocks can improve block-cache hit rate by exploiting spatial locality. At the same time, they are likely to increase miss latency, which depends on decompression latency, memory fetch latency, and read amplification.

Decompression emits cache lines sequentially, so cache lines in larger blocks have a higher decompression latency. Both the maximum latency, the time it takes to decompress the last cache line in the block, and the mean latency, the

time it takes to decompress the midpoint cache line in the block (assuming a uniform access distribution across the cache lines), increase with block size. Similarly, larger blocks incur higher fetch latency because it increases with the amount of data fetched per cache miss (*i.e.*, the size of the compressed block). Although larger blocks compress better, the gain is not enough to offset the increased raw block size.

Read amplification occurs when the memory expander decompresses an entire block to serve a single cache line. In the worst case, if only a single 64B cache line is needed from a 4KiB block, there can be up to a $\sim 64\times$ read amplification. Large bursts of cache misses can saturate the DRAM bandwidth of the expander and introduce queuing delay. Smaller blocks reduce read amplification, but larger blocks may actually reduce overall memory reads if many cache lines are accessed within the decompressed block. Therefore, whether read amplification translates into large latency depends on the access patterns to the memory expander.

Compression algorithm tradeoffs. Memory expanders present different algorithm tradeoffs than software-based far memory systems such as Google Zswap [9]. Software approaches use the simplest algorithms to minimize the already large latency of software decompression and the address indirection layer, which relies on page faults to intercept accesses to compressed memory. Memory expanders, in contrast, decompress in hardware through a direct load/store interface, where decompression and memory fetch latency operate at a similar scale. This changes the tradeoff: more complex algorithms increase decompression latency, but also provide comparable latency benefits as more compressed blocks reduce memory fetch latency and queuing delay.

C. Open Questions

Hardware compression has complex tradeoffs, but expanders today default to safe and well-understood choices inherited from software systems. The OCP specification sets LZ4 as the baseline, and early CXL silicon from Marvell and ZeroPoint implements only simple compression [6], [7]. The specification and commercial devices propose 4KiB and 1KiB block sizes, but offer no guidance on how these affect performance. The next sections provide an in-depth analysis of how expanders perform at hyperscale.

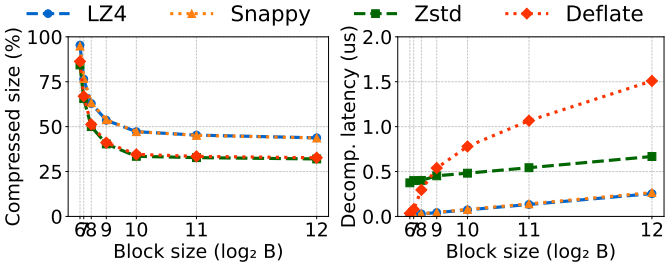
III. Compression at Sub-Page Granularity

This section quantifies the raw compression performance across block sizes and algorithms. Finding the right configuration requires understanding two more factors: (1) the compressibility of cold pages at sub-page granularity and (2) the locality of accesses to cold pages. This section examines compression configurations that current architectures already support or can adopt with minimal hardware changes, showing how these configurations alter host performance.

A. Methodology

We use HyperCompressBench [22] to generate representative expander data. Measuring the compressibility of different algorithms on live production data imposes an unacceptable privacy risk. Consulting with datacenter operators, they recommended using HyperCompressBench as it is sufficiently representative: its average compressibility is that observed under production zswap-based far memory [9]. Furthermore, Section IV-E shows the results depend on this average rather than on the detailed distribution or per-block assignment. We measure the compression size and latency for seven block sizes (64B-4KiB, in powers of 2). We cap block size at 4KiB because it is the largest unit at which the OS guarantees physical contiguity.

We use four compression algorithms: LZ4 and Snappy, which are optimized for low latency, and Zstd and Deflate, which target higher compression ratios. These four algorithms span the range from low-latency to high-ratio general-purpose compression and are the main candidates that the industry is considering for CXL memory expanders. We measure hardware decompression latency using FPGA-based Vitis compression kernels at a 300 MHz clock [23] and derive an FPGA-to-ASIC scaling factor by comparing them against an ASIC-based IAA and QAT decompressors [24], [25], which also aligns with frequency scaling.



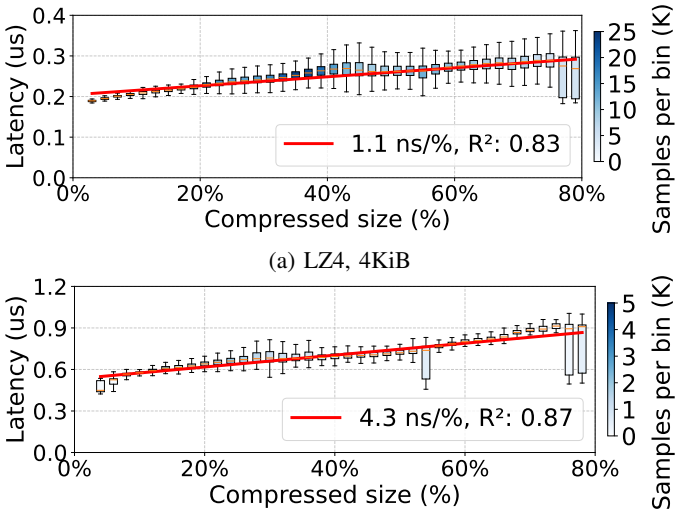
(a) Compressed size (b) Decompression latency

Fig. 2: Average compressed size and decompression latency of the HyperCompressBench corpus. Zstd shows 2.5x higher decompression latencies and a 13-37% reduction in compression size than LZ4. Larger blocks yield diminishing compression size reductions beyond 1KiB.

B. Raw Performance Analysis

Figure 2 shows that Zstd and Deflate reduce compression size over LZ4 and Snappy by 13-37%. Algorithms with the same computational foundations yield similar compressed sizes. All algorithms compress larger blocks more, but there are diminishing returns after 1KiB. Tiny blocks see small space savings. More complex algorithms have higher latencies. Zstd has a fixed minimum latency of 370ns, which is more than an order of magnitude higher than the others for small blocks. Deflate's latency for tiny blocks is small and competitive with LZ4 and Snappy. For larger blocks, Deflate and Zstd have $>2.5\times$ higher latency than LZ4 and Snappy.

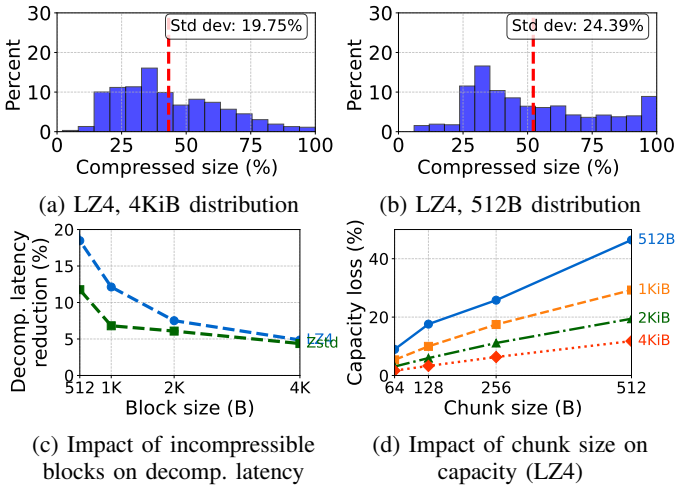
Figure 3 shows a highly linear relationship between compressed size and decompression latency. We omit Snappy and Deflate from the figure due to space limitations; Deflate shows



(a) LZ4, 4KiB

(b) Zstd, 4KiB

Fig. 3: Linear relationship between decompression latency and compressed size. Complex algorithms (*e.g.*, Zstd) show a steeper slope than simple algorithms (*e.g.*, LZ4).



(a) LZ4, 4KiB distribution

(b) LZ4, 512B distribution

(c) Impact of incompressible blocks on decomp. latency

(d) Impact of chunk size on capacity (LZ4)

Fig. 4: Compressed-size distributions and their impact on latency and capacity. Smaller blocks and simple algorithms yield more incompressible blocks, reducing latency. Larger chunks and smaller blocks waste more space.

a 27× steeper slope than LZ4. Blocks that take the longest to decompress save the least capacity.

Memory expanders exploit this linearity by bypassing decompression for *incompressible blocks*: if a block's compressed size exceeds 80% of its original, the expander stores it uncompressed. This effectively cuts average and worst-case latency at a small cost in capacity, especially for algorithms like Deflate with steep slopes.

C. Performance in the Memory Expander

As shown with incompressible blocks, architectural details create a gap between raw compression results and the capacity and latency the host actually experiences. Moreover, the magnitude of this gap varies across compression configurations. Incompressible blocks are more effective for smaller blocks and simpler algorithms (Figure 4c). Simpler algorithms and smaller blocks produce more incompressible blocks, amplifying the latency benefit of bypassing decompression. For LZ4, the incompressibility rate increases from 4.8% at 4KiB to 18.5% at 512B (Figure 4a–4b), translating into an 18% decompression latency reduction.

Another critical architectural property is *chunk quantization*. Expanders manage memory in fixed-size chunks. Compressed blocks are stored in these chunks, lowering effective capacity through internal fragmentation (Figure 4d). Using smaller blocks wastes more space, demanding fine-grain memory management. The distribution of compressed sizes also affects the capacity wasted. Figure 4b depicts the distribution unfavorable to chunk quantization, where 27% of the blocks compress to just over 128B (two right bars of 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.

While the analysis so far can be determined from compression results alone, three runtime properties explain why large decompression latency does not proportionally increase 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, as decompressed blocks are cached, the subsequent accesses bypass decompression. Third, memory fetches themselves incur substantial latency that depends on queuing. More complex algorithms that fetch less data enqueue fewer requests at the memory controller. Therefore, a detailed understanding of the capacity–latency tradeoff requires accounting for access patterns.

IV. Quantifying Tradeoffs at Hyperscale

This section evaluates the end-to-end performance of compression configurations using cold memory traces from a hyperscaler fleet.

A. Methodology

We use cold memory traces obtained from a far memory system deployed in a hyperscaler fleet (equivalent to Google's [26]). Each trace records the physical addresses and timestamps of accesses to cold pages: cache and host DRAM accesses are excluded. We have ten traces, Trace-0 through Trace-9, each collected from a randomly selected server running a very different mix of hundreds of concurrent workloads.

We augment these traces by assigning every block a (compressed size, decompression latency) pair sampled from decompression results of HyperCompressBench in Section III. Jointly gathering access traces with compressibility information imposes unacceptable performance overhead and privacy risks on production servers. We rely on HyperCompressBench for compressibility for the same reasons outlined in Section III-A.

TABLE I: 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). LZ4/512B achieves the peak performance, while Zstd/2KiB balances improvements in both metrics.

	LZ4		Zstd	
Block size	Access lat. reduction	Capacity increase	Access lat. 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%

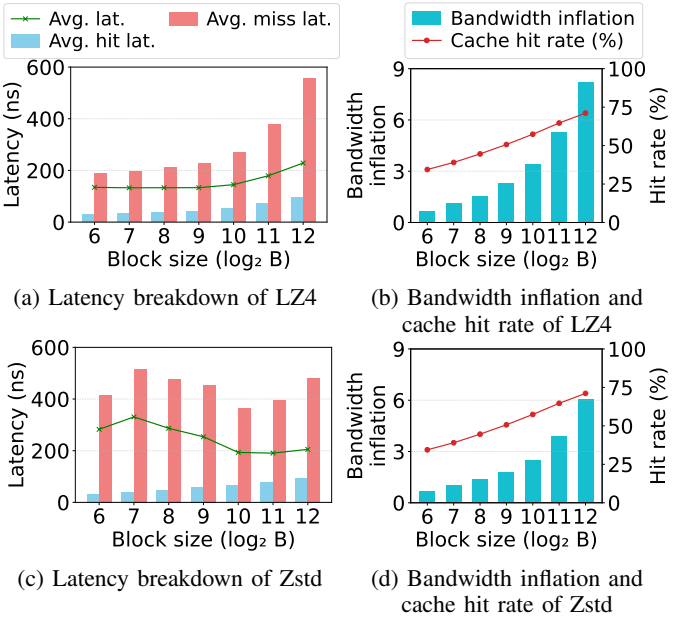
Our experimental setup reflects the near-term deployment target for the hyperscaler: repurposing DDR4 and minimal hardware changes to the proposed architecture. As Section III suggests, Snappy and LZ4 can be used interchangeably, and Zstd is faster than Deflate at block sizes that offer sufficient compression. We therefore focus on LZ4 and Zstd, both of which can be implemented at reasonable hardware cost [22], and the OCP specification explicitly provisions them [5].

We model a commercially representative memory expander on gem5, configured to match the Marvell Structera device [6] and the OCP specification [5]: four 8GB DDR4-2600 DIMMs with one DIMM per channel, a 16MB 8-way set-associative LRU block cache with 30ns access latency, and sufficient decompression engines to saturate the full DRAM bandwidth. The decompression engine uses the compressibility assigned to each block: the compressed size sets the number of bytes fetched from DRAM, and the decompression latency sets the service time for that block. We validate that the simulation closely matches the idle latency of the actual device [6]: cache hit (0.00%), incompressible block (2.59%), and compressed block (-3.59%).

B. Optimal Compression Configuration

Table I 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% lower), at the cost of a 24.4% smaller effective capacity than the baseline.

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. 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 simultaneously increases capacity and reduces latency.



(a) Latency breakdown of LZ4

(b) Bandwidth inflation and cache hit rate of LZ4

(c) Latency breakdown of Zstd

(d) Bandwidth inflation and cache hit rate of Zstd

Fig. 5: Detailed latency and read amplification breakdown of LZ4 and Zstd on a representative trace, Trace-3. Hit and miss latency grow as block size increases from 512B to 4KiB. While the cache hit rate is identical in both compression algorithms, Zstd compresses more and reads less from memory, resulting in lower bandwidth inflation.

TABLE II: P95/P99 tail latencies (μs) across compression configurations. Tail latency increases more rapidly with block size in LZ4 than in Zstd.

	256B	512B	1KiB	2KiB	4KiB
LZ4	0.27/0.53	0.30/0.55	0.37/0.58	0.54/0.74	1.42/1.81
Zstd	0.71/0.86	0.79/0.94	0.81/0.94	1.04/1.11	1.22/1.46

C. Bandwidth Inflation, Queuing, and Delayed Hits

To explain the trends across compression configurations, Figure 5 decomposes access latency into hit latency, miss latency, and cache hit rate. The miss latency is further divided into decompression, memory fetch, and queuing. The first two components are proportional to the compressed block size. To better reason about memory queuing, we define *bandwidth inflation* as the ratio of the total data read from DRAM to the total data the host actually requests. While read amplification measures a single cache miss, bandwidth inflation accounts for multiple hits served from a single decompressed block. If every cache line in a decompressed block were accessed, bandwidth inflation could fall below 1. Configurations with high bandwidth inflation are therefore more susceptible to memory queuing delays.

For block sizes 1KiB to 4KiB, Zstd shows lower access latency than LZ4. Cache hit rates and hit latency are similar between the two algorithms, so the difference comes from miss latency. LZ4 decompresses faster than Zstd, but its higher bandwidth inflation creates larger queuing delays that more

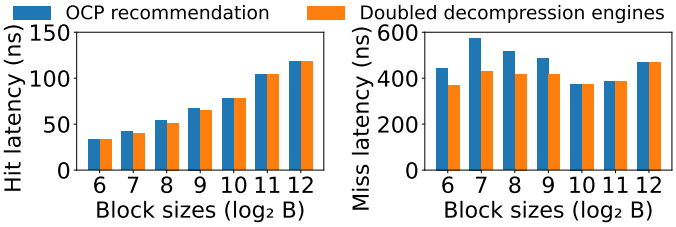


Fig. 6: Average hit and miss latency with recommended and doubled number of decompression engines for Zstd. Doubling the engine count reduces the latency for small block sizes.

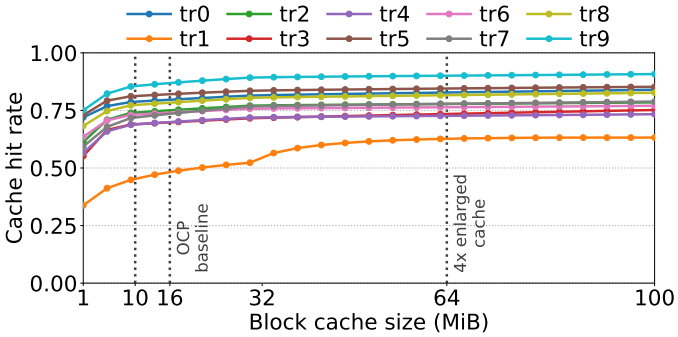


Fig. 7: Cache hit rate vs. block cache size (1 MiB–100 MiB). Hit rate plateaus beyond 10MiB.

than offset this advantage. For example, on average, LZ4 compresses a 4KiB block to 44.5% of its original size, with read amplification of 28× (44.5% × 4KiB / 64B). With an average block cache hit rate of 71%, 3.4 cache lines are served from a 4KiB block, resulting in bandwidth inflation of $28 \times (1 - 71\%)$=8.12. Due to its better compression, Zstd with the same block size has a bandwidth inflation of 6.0. The difference in tail latency is even larger; Table II shows that LZ4 with 4KiB has 63% and 24% higher P99 latency than Zstd with 2KiB and 4KiB blocks, respectively.

For LZ4, access latency grows with the block size. Larger blocks achieve higher cache hit rates, but both miss and hit latency grow faster. Miss latency increases because larger blocks have longer decompression and fetch latency. Bandwidth inflation also rises with block size, despite better compression and higher hit rates. Hit latency increases due to *delayed hits* [27]: when a request arrives for a block already being decompressed, it counts as a hit, but its latency includes the remaining decompression time. Smaller blocks see fewer delayed hits because it is less likely that a future access will land during the shorter decompression time.

Zstd has a more complex latency trend across block sizes. For blocks larger than 1KiB, miss latency grows primarily due to memory queuing delays. For blocks smaller than 512B, 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 queuing 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.

As the smaller blocks are compute-bound, the OCP guideline of matching decompression throughput with memory bandwidth is insufficient. Matching average throughput does not prevent queuing; decompression time varies with data content, and memory requests can arrive out of order. Figure 6 confirms this: doubling the number of decompression engines for Zstd reduces miss and hit latency for small blocks, as queuing on decompression is reduced.

D. End-to-End Analysis

Read amplification translates every cache miss into a full block read from DRAM, making memory expanders particularly sensitive to bandwidth pressure. This makes memory bandwidth efficiency the key differentiator between compression algorithms. Zstd's better compression not only provides greater capacity but also lower latency than LZ4 by reducing the memory queuing delay. 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, memory bandwidth efficiency is more important than computational efficiency.

The computational complexity of Zstd results in a 36% increase in size, and 76% increase in power over LZ4 with the same throughput [28]. The cost overhead is negligible compared to the substantial memory Zstd saves. Moreover, this is a tiny portion (<1%) of the controller [7], [19], which is dominated by the SRAM block cache and CXL/DDR PHYs.

E. Sensitivity Study

Cache size. Figure 7 shows that cache sizes beyond 10MiB exhibit diminishing returns in hit rates for the fleet traces, so the latency benefit of Zstd over LZ4 remains consistent in that region. Even with a 4x enlarged cache, the cache hit rate increases by 4.4% on average. Zstd with 4KiB and 2KiB still provides 5% and 7% lower access latency, respectively.

Compressibility. Zstd's latency advantage is driven primarily by its higher average compression ratio, not by the detailed distribution or by how compressibility couples to access patterns. With two distributions with smaller or larger deviation and two biased couplings in which high-miss blocks compress more or less, Zstd with 2KiB blocks remains 8.5–19% faster than LZ4 with 4KiB blocks. The distribution has little effect because the compressed-size-to-decompression-latency slope is gentle for both algorithms (Figure 3). The coupling has little effect because over 95% of blocks are accessed fewer than 10 times, leaving few high-miss blocks for any bias to act on.

DRAM configurations. Increasing DRAM bandwidth would reduce the relative benefit of Zstd, but the evaluated configuration already represents the practical upper end of a

TABLE III: Benchmarks widely used in tiered memory research.

Benchmark	Description
TPC-C [30] on Silo [31]	OLTP workload on the Silo in-memory database engine.
YCSB [32] on Redis [33]	Cloud-serving workload on the Redis in-memory key-value store.
GAPBS [34]	PageRank on Twitter graph.
Liblinear [35]	Linear classification on KDD12 dataset.
`603.bwaves`	Explosion modeling (SPEC CPU2017) [36].
`654.roms`	Ocean modeling (SPEC CPU2017) [36].
Bench. mix	A mixture of all the above.

cost-driven cold tier. Four DDR4 channels already match the highest channel count found in commercial CXL expanders. Eliminating DRAM-side queuing under the 5–10× bandwidth inflation we observe would require memory bandwidth several times higher than the host-link bandwidth, implying roughly an order of magnitude more DDR4 channels. Such configurations are constrained by die area and cost. This bandwidth-bound regime is therefore inherent to the target cold tier and is precisely where Zstd's higher compression efficiency provides its benefit.

F. Representing the Diversity

The fleet traces are essential to ground analysis in real behavior, but they represent a small subset of the diversity at hyperscale. No tractable trace collection can fully cover the full diversity: datacenters have millions of servers running thousands of workload types. Each trace is tied to a single system configuration, tiering policy, and workload mix, all of which evolve over time. Moreover, sharing raw traces is prohibitive due to privacy and proprietary concerns [29].

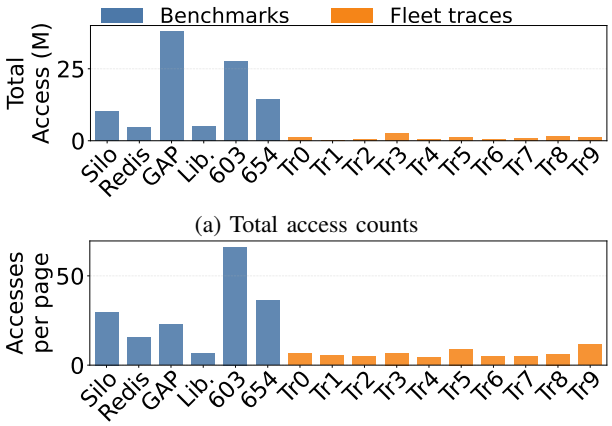
A natural alternative is benchmarks, which offer configurable and reproducible workloads. However, we find that widely used tiered memory benchmarks reach different design conclusions (methodology details in Section V). All benchmarks favored LZ4 over Zstd in terms of latency—the opposite of our fleet trace results. The reason is that benchmarks generate far less memory traffic: block cache hit rates consistently exceed 85%, keeping bandwidth pressure low enough that LZ4's bandwidth inflation never exceeds 5. Under these conditions, LZ4's faster decompression dominates, and the bandwidth advantage of Zstd never materializes.

V. Benchmark Representativeness

This section analyzes why benchmarks differ so markedly from fleet traces and why relying on them for tiered memory evaluation leads to misleading conclusions.

Methodology. Table III summarizes the benchmark suite. We run benchmarks on the system using a CXL memory as a cold tier. Cold pages are detected and migrated to the CXL tier based on the AutoNUMA-tiering [37]. We then collect only the accesses that reach the cold tier via CXL-Tracer [38].

Access counts. Figure 8 shows that benchmarks place substantially heavier load on the cold memory than the fleet, 15× in total accesses and 4.5× in accesses per page. Given that each fleet trace aggregates cold memory from hundreds of co-running workloads, this order-of-magnitude higher accesses in benchmarks indicates that benchmarks behave as extreme stress tests. The per-page access count distributions also differ. Fleet traces are dominated by pages touched only a few times, and the number of pages drops sharply as the access count increases. At the same time, fleet traces contain a small number of pages accessed highly frequently. In contrast, benchmarks show a gradual decrease in the distribution and do not exhibit such extreme outliers.



(a) Total access counts

(b) Average access counts per page

Fig. 8: Access counts in benchmarks and fleet traces. Benchmarks generate far more cold memory accesses.

These differences may overstate the performance benefits of memory tiering technologies. As benchmarks touch cold memory more frequently, the performance penalty of the slow tier becomes larger. This makes tiering mechanisms designed to hide that penalty appear more valuable than they would under production-like cold memory access patterns.

Locality. Figure 9 projects benchmarks and fleet traces into a 2D locality space using the locality model we introduce in Section VI. A higher PCA value corresponds to higher locality. The two groups show distinct clustering, confirming that their cold memory locality is fundamentally different. Benchmarks show higher temporal locality than fleet traces running hundreds of concurrent workloads. Along the spatial axis, benchmarks exhibit denser within-page access patterns resembling sequential scans, while fleet traces are sparser, reflecting pointer-chasing behavior. Co-running multiple benchmarks does not bridge the gap: the combined trace averages the individual localities while inflating access counts.

These differences can bias design conclusions. Benchmarks overstate the benefit of coarser-grained migrations [39], whose transfer cost is easily amortized by dense access patterns. Moreover, benchmarks tend to favor frequency-based migration policies because their pages are accessed more often, whereas production workloads with higher access sparsity are more likely to favor recency-based policies [40], [41].

Conclusion. Existing benchmarks are poor proxies for produc-

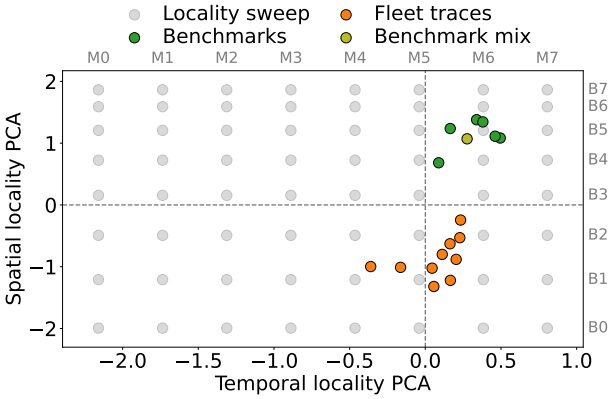
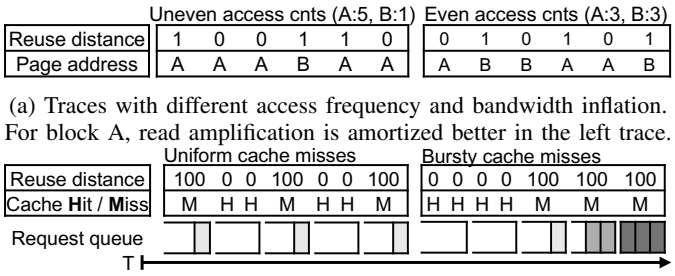


Fig. 9: Mapping traces into 2D PCA space. Benchmarks and fleet traces show distinct clustering.



(a) Traces with different access frequency and bandwidth inflation. For block A, read amplification is amortized better in the left trace.

(b) Traces with different cache miss ordering and queuing delay. Cache misses are clustered in the right trace, resulting in higher queuing delay.

Fig. 10: Traces with the same reuse distance distribution having very different memory expander performance.

tion cold memory and cannot predict real system performance. Fleet traces are scarce and proprietary. This motivates a model-driven approach that captures only what matters about a trace—its locality—and generates new traces with different locality on demand.

VI. Modeling Cold Memory Locality

In this section, we present the first part of *COLA*, a parameterized model of **CO**ld memory **L**oc**A**lity. Later in Section VII, we introduce the second part of *COLA*, generating traces from these parameters.

A. Why Parameterize Locality?

COLA needs to achieve two goals: (1) represent the locality of observed traces and (2) generate traces from the specified locality. This bidirectional mapping—trace to locality and locality to trace—requires encoding locality space into a *small set* of parameters with *interpretable semantics*. The parameter set must be small because *COLA* is intended to be a human-controllable design-space exploration tool. At the same time, the parameterization must retain explicit interpretable semantics to provide architects with knobs that can be adjusted with a clear intent.

Such parameterization enables architects to accurately predict the performance implications of different access patterns. By mapping traces from any source—fleet, benchmarks, and synthetic—into the same space, architects can quantify how traces differ and anticipate how those differences lead to different design conclusions. This is precisely what makes the benchmark–fleet comparison in Section V possible.

Moreover, the parameterization enables systematic "what if" exploration. By interpolating between observed traces or extrapolating to unobserved regimes, architects can ask "what if cold memory had different locality?" without collecting new traces. Figure 9 demonstrates this expressiveness: sweeping parameters across plausible extremes produces a grid that encloses both the benchmark and fleet clusters and extends well beyond them.

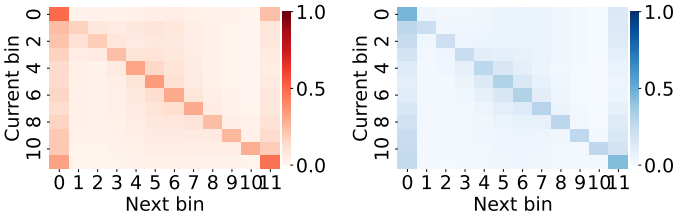
The remainder of this section describes which locality profiles matter for tiered memory systems, and how *COLA* reduces each profile into a compact set of parameters.

B. Rethinking Locality for Tiered Memory

Motivated by the findings in Section IV-C—bandwidth inflation and queuing are critical to access latency—we identify three locality profiles that shape memory expander performance. Bandwidth inflation depends on *access frequency*, how many cachelines are served per decompressed block. Blocks accessed many times amortize read amplification far better than those touched once. Whether the large bandwidth inflation translates into queuing delay depends not only on the cache miss rates but also on how they are *temporally clustered*. Workloads that have bursts of misses and bursts of hits will perform worse than workloads where misses are distributed evenly. Lastly, the above profiles must be understood at *multiple block size granularities*.

These locality profiles matter beyond memory expanders, in tiered memory systems broadly—including page migration, placement, and prefetching [42]–[45]. Access frequency drives placement decisions: frequently accessed pages benefit from fast-tier residency, while sparsely accessed pages waste migration bandwidth [42], [43]. Tiered memory systems often exhibit multiple latency regimes, for example, in RDMA-based disaggregated memory with local caches or slower tiers with on-device caching [5], [46]–[50]. Temporal clustering of long-latency accesses can build queues on the critical path and disproportionately degrade performance. Multi-granular locality is inherent to tiered memory, because systems manage memory at larger units (pages) than access granularity (cache lines).

Existing locality models, however, are built around reuse distance distributions [51]–[53] to target general memory access patterns. Reuse distance measures, for a given cache line C, the number N of distinct cache lines accessed between two accesses to C. Reuse distance can directly model a least-recently-used (LRU) cache replacement policy: if N is greater than the size of the cache, it misses.



(a) Original Markov matrix (b) Reduced Markov matrix

Fig. 11: Markov transition matrix of Trace-0 and its low-parameter reconstruction. Dominant peaks are preserved in the reconstructed matrix, and the RMS error is marginal (0.016).

A reuse distance distribution can accurately predict LRU cache miss rates, but it fails to capture the locality profiles essential to tiered memory. It does not record which block each reuse distance belongs to, so traces with identical reuse distance distributions but different access frequency skew yield different bandwidth inflation (Figure 10a). Nor does it preserve the ordering of cache misses; Figure 10b shows two traces with the same reuse distance distribution but different miss orderings, producing very different queuing behavior.

None of the prior works [15]–[17], [54] models temporal clustering of cache misses or per-block access frequency—precisely the profiles that dominate performance in tiered memory systems. WEST [16] is tied to a fixed 64B cacheline granularity. STM and HALO [15], [54] model multi-granular locality but are not amenable to parameterization or trace synthesis. Lightweight generators like Apex-MAP [55] are inadequate for replicating the fine-grained access characteristics of real system workloads.

C. Reuse Distance Clustering

We model the reuse distance sequence as a first-order Markov chain. There are 12 states. Bin 0 represents reuse distance $r = 0$, and bin 11 represents $r = \infty$, i.e., the first access to a block. For finite $r > 0$, we use two logarithmic bins per decade. Bin b covers $10^{(b-1)/2} \leq r < 10^{b/2}$ for $1 \leq b \leq 10$. For example, bin 4 corresponds to reuse distances 32–99. We use a logarithmic scale because smaller reuse distances are more critical to cache performance. Each state has a probability distribution of transitioning to other states. This model can capture loops and other burst behaviors.

Figure 11a shows an example reuse distance matrix, from Trace-0, representative of all of the traces. Density concentrates similarly across all fleet and benchmark traces: the bin-0 column (consecutive accesses to the same block), the bin-11 column (accesses to new blocks), and the diagonal (a stream across a working set at a repeating reuse distance).

The full matrix is too fine-grained for a simple parameterization. We fit each matrix with a sum of seven 2D Gaussians aligned with its recurring structure: one each for the bin-0 column (high-locality bursts), the bin-11 column (new-page accesses), and the diagonal (looping), plus four fixed corners. This yields only 24 parameters: 3 axis positions, 7 amplitudes, and 14 sigmas. Figure 11b shows the reconstruction for Trace-0 using these reduced parameters, with an RMS error of 0.016 (0.018 averaged across all traces).

The parameterization enables adjusting the locality with clear architectural intents. For example, increasing the amplitude of (11, 11) centered Gaussian means that cold misses will arrive in longer bursts. A purely statistical reduction, such as PCA, could reduce the parameters further, but its components carry no physical meaning and would not yield the interpretable parameterization we pursue (Section VI-A).

D. Block Access Frequency

The Markov chain models the sequence of reuse distance bins, but not which specific block each access targets. For example, when the Markov chain samples bin 4 (reuse distance 32–99), we have 68 candidate blocks to access within this range, and which one is chosen shapes each block's total access count. This choice matters because real traces exhibit highly skewed access frequencies: most blocks are touched only a few times, while a handful are accessed thousands of times. Naively picking a block—for example, uniformly at random within the bin—destroys this skew. We instead record the observed access frequency distribution and use it to guide block selection during trace synthesis (Section VII).

The access frequency distribution has a long tail whose exact shape varies across traces, ruling out a fixed parametric form such as an exponential. *COLA* partitions access counts into logarithmically scaled intervals and records the number of pages and total accesses per interval, then reconstructs the full distribution via an integer linear program (ILP).

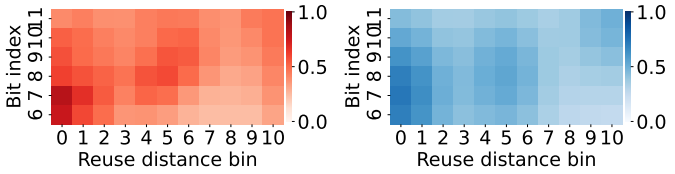
E. Multi-granular Locality

Multi-granular locality captures how block granularity affects reuse distance (Section VI-C) and access frequency (Section VI-D). Therefore, multi-granular locality captures how quickly the cache hit rate drops with shrinking block size (*i.e.*, spatial locality): two accesses to the same 4KiB block may land in different 2KiB sub-blocks.

The effect of block granularity is related to how sparsely each block is accessed. If the access location within the block is uniformly distributed, then cutting the block size in half would at least double the reuse distance and halve the access frequency. In contrast, if accesses are concentrated on a single cache line, reducing the block size may leave both reuse distance and access frequency nearly unchanged.

A brute-force approach would compute a Markov matrix and access frequency distribution for every block size. This multiplies the parameter count by 7, making the locality space impractically large to explore. More critically, synthesizing a trace that simultaneously satisfies all seven sets of constraints is computationally intractable.

COLA takes a simpler statistical approach, calculating the Markov matrix and access frequency only at the highest granularity (4KiB) and then measuring the intra-page access sparsity. The key question is: when two accesses hit the same 4KiB block, do they also land in the same 2KiB sub-block?



(a) Original bit flip matrix (b) Reduced bit flip matrix

Fig. 12: Bit flip matrix of Trace-2 and its low-parameter reconstruction. Dominant peaks are preserved in the reconstructed matrix, and the RMS error is marginal (0.027).

This is equivalent to asking whether the address bit that distinguishes the two halves (bit 11) flips between the same page accesses. A bit flip matrix records this probability for each combination of reuse distance bin and address bit. For example, the entry at reuse distance bin 4 and bit 11 gives the probability that, among accesses to the same page spaced 32–99 accesses apart, the 11th bit flips. A low probability means accesses are spatially concentrated; a high probability means they are spread across sub-blocks.

Figure 12 shows the bit flip matrix for Trace-2. All of the fleet traces are similar to Trace-2. These matrices share a common structure: a peak at low reuse distances and low bit indices (strided scans), a broad spread at high reuse distances (diverse access patterns across sub-blocks), and a smaller peak in the middle (large loops). As with the Markov matrix, we represent each as a sum of three 2D Gaussian mixtures aligned with these structures: four parameters each (location and two amplitudes) plus four shared sigmas, 16 numbers in total. Figure 12b shows the reconstruction for Trace-2, with an RMS error of 0.027 (0.040 averaged across all traces).

F. Inter-access Time

The final parameters of *COLA* capture access timing. Access intensity directly affects queuing pressure on the memory expander, and real traces alternate between bursts of rapid accesses and quieter intervals. A single uniform inter-arrival time cannot capture this behavior. *COLA* models arrival times with two Poisson distributions: one for intra-burst arrivals and one for inter-burst gaps. *COLA* then fits a quadratic function that maps each reuse distance bin to the probability that the access belongs to a burst. This captures the correlation between reuse distance and burstiness: repeated accesses to the same block (bin 0) are likely to arrive in bursts.

VII. *COLA* Trace Generator

COLA is backed by a trace generator that synthesizes traces from any point in the locality space, enabling actual evaluation. This section describes the generation process and evaluates how faithfully synthesized traces reproduce the locality and performance characteristics of the original traces.

A. Synthesizing Locality-Representative Traces

COLA synthesizes a trace in three steps: generate a page access trace, determine the offsets within the pages, and then assign timestamps to the accesses.

The first stage uses the Markov matrix and page access frequency distribution. It begins by assigning each page in the working set a target access count sampled from the distribution. The sequence begins with an infinite reuse distance access to page 0. For each subsequent access, the generator samples the next reuse distance bin from the Markov matrix. The range of reuse distances determines the candidate blocks from the earlier part of the trace. The generator then selects one of these blocks to match the target access count. For example, if the next reuse distance bin is 4, with reuse distances 32-99, the generator selects from those 68 blocks based on which block's access count differs most from its assigned target access count. If there is no valid page (this occurs at the start of a trace, when not all reuse distances exist yet), it switches to an infinite reuse distance, representing the cold miss to a new page. This stage is terminated when the number of pages exceeds the working set size.

The second stage assigns page offsets to the page trace from the first stage using the bit flip matrix. Starting from the page trace, the generator decides for each access whether the most significant offset bit (bit 11) flips compared to the most recent access to the same page, based on the bit flip probability for that access's reuse distance. This splits a 4KiB block into 2KiB blocks and yields a 2KiB trace. The process repeats recursively—deciding bit 10 to produce a 1KiB-level trace and so on—until the 64B trace is constructed.

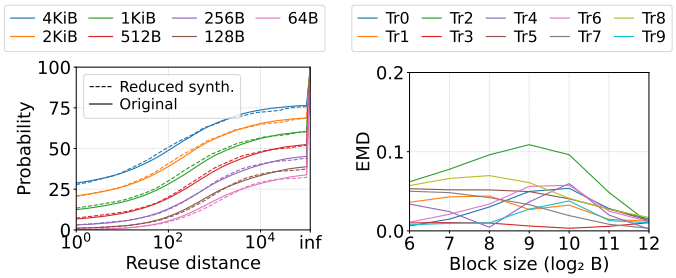
Lastly, timestamps are assigned by sampling from Poisson distributions. For each access, the model uses its reuse distance and computes the probability that the next access will be soon (burst Poisson) or later (interval Poisson). It samples a time interval from the selected Poisson distribution.

B. Evaluation

Each fleet trace is reduced to a few parameters used to synthesize fleet-like traces (released with *COLA*). By design, the synthesized traces differ from the originals; an exact copy would expose proprietary content. *COLA* instead reproduces only the locality that governs cold-tier evaluation.

In this subsection, we compare the fleet traces with two types of synthesized traces: those using the full Markov and bit flip matrices, and those using reduced parameterization.

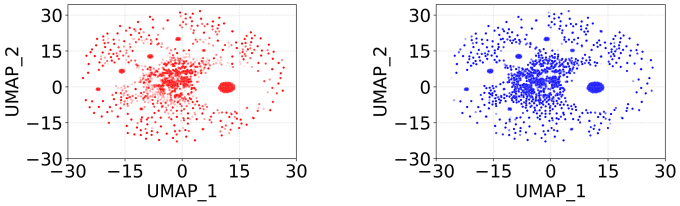
Locality analysis. Figure 13 shows that the reduced synthesized traces closely replicate the temporal locality of the original traces. We evaluate the similarity by measuring the Earth Mover's Distance (EMD) between the reuse distance cumulative density functions (CDFs), which measures the minimum "work" required to transform one distribution into another, normalized to the maximum distance. For example, the reuse distance CDFs of Trace-3 are shown in Figure 13a, corresponding to the lowest average EMD observed in Figure 13b. The Markov matrix implicitly defines the page-level reuse distance distribution, so all traces show the smallest EMD with a 4KiB block. The bit flip matrix accurately captures the increase in reuse distance as block size decreases, which shows as a right shift of the CDFs in Figure 13a. The average EMD for each trace is 0.008–0.07, and 0.03 on



(a) Reuse distance CDFs comparison of Trace-3

(b) Earth mover's distance between reuse distance CDFs

Fig. 13: Similarity comparison between the reuse distance CDF of the original trace and its reduced synthesis counterpart. The average EMD for each trace is 0.008–0.07.



(a) Original Trace-0 projection

(b) Reduced-synthesized Trace-0 projection

Fig. 14: UMAP projection of per-page reuse distance distribution vectors to 2D space. *COLA* captures not only the reuse distance distribution but also how reuse distances are associated with memory addresses.

average. The high similarity suggests that the cache hit rates are nearly identical between the original and synthesized traces across block sizes and cache capacities.

As we use access frequencies to decide the accessed blocks among the reuse distance bins, this helps *COLA* capture how reuse distances are associated with memory addresses. This synergetic effect offsets the loss of precision introduced by quantized reuse distance bins. For each page, we collect the reuse distances of its accesses as a vector and project these vectors into two dimensions using Uniform Manifold Approximation and Projection (UMAP). As illustrated in Figure 14, the projections of these vectors are highly consistent between the original and synthesized traces.

Access frequency and the bit flip matrix together determine the intra-page address distribution or the footprint: how many distinct sub-blocks within each page are accessed. For example, heavily-accessed pages with high bit-flip ratios yield more distinct block accesses per page. Figure 15 shows that synthesized traces closely match the original footprints, with average and maximum errors of 4% and 15%.

Expander performance. Beyond locality metrics, we evaluate whether *COLA* reproduces the end-to-end performance of memory expanders. Figure 16 compares cache hit rates and access latencies between original and synthesized traces for three representative cases (smallest, median, and largest cache

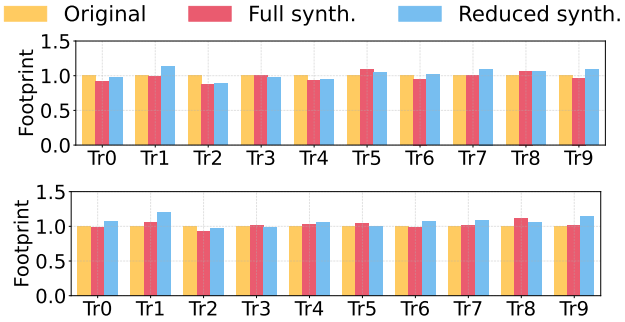
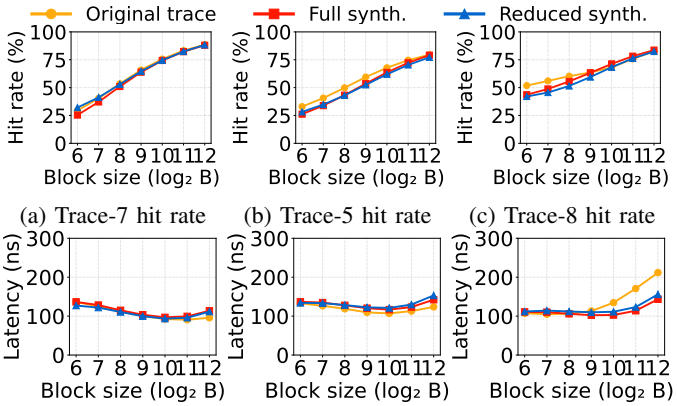


Fig. 15: Footprint of traces with 512B (above) and 64B (below) block sizes, normalized to original traces. The number of distinct cache lines in the full synthesis differs by only 7% from the original.



(a) Trace-7 hit rate

(b) Trace-5 hit rate

(c) Trace-8 hit rate

(d) Trace-7 latency

(e) Trace-5 latency

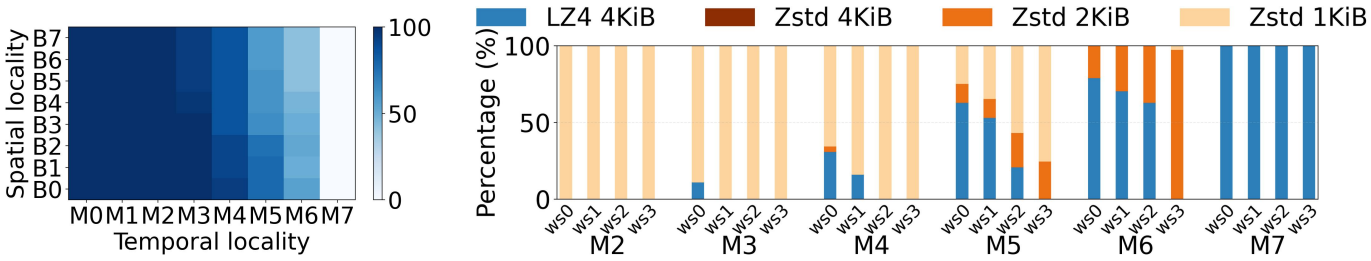
(f) Trace-8 latency

Fig. 16: Simulation results of synthetic traces with LZ4 compression. Trace-7, 5, and 8 show the smallest, median, and largest cache hit rate errors, respectively. *COLA* faithfully reproduces the locality for tiered memory proposed in VI-B, and thus the expander performance.

hit rate error). Across all traces, reduced synthesis achieves an average absolute cache hit rate error of 3 pp and predicts access latency within 6.1%–21% (9% on average). Larger latency error is attributed to the actual access patterns being more or less bursty than what our timestamp method can capture. Nevertheless, latency trends are consistent across all traces, and the synthesized traces identify the same optimal block size and algorithm as the originals.

VIII. Generalization

This section uses *COLA* to comprehensively study whether the results in Section IV-D hold across a wide range of access patterns demonstrated by *COLA* parameter sweep (Figure 9). We generate traces by varying five parameters: working set size (0.5×, 1×, 2×, 4× the fleet mean), temporal locality (M0–M7), spatial locality (B0–B7), access count distribution (most frequent count at 1, 10, or 50), and burstiness (three levels). Temporal locality spans from 10% to 90% cache hit rates, and spatial locality spans from an average bit flip rate



(a) Percentage of traces where Zstd achieves lower latency than the baseline (LZ4 4KiB).

(b) Composition of the lowest-latency configuration across working-set size (ws0–3, increasing) and temporal locality. Larger working sets prefer Zstd and a smaller block size.

Fig. 17: Sensitivity results on *COLA* parameters. The latency benefit of Zstd over LZ4 applies to 75% of traces. Larger numeric annotations indicate a higher degree of the corresponding metrics. Traces with lower spatial or temporal locality and larger working set size prefer Zstd over LZ4.

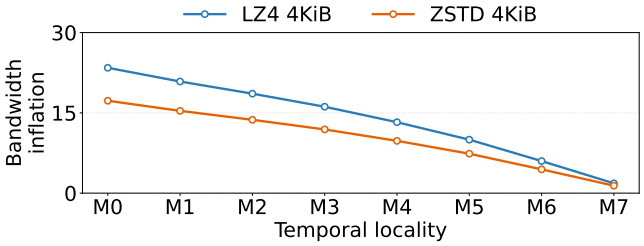


Fig. 18: Bandwidth inflation of LZ4 and Zstd with 4KiB blocks across temporal locality levels. As cache hit rates decrease (lower temporal locality), the bandwidth inflation gap between the two compression algorithms widens.

of 0.2 to 0.8. The fleet traces fall around the medium level of each. We evaluate 2,304 synthetic traces under two algorithms (LZ4 and Zstd) and three block sizes (1KiB–4KiB).

Zstd outperforms the baseline for both capacity and latency for 75% of the generated traces. Figure 17a shows that LZ4 achieves lower latency only when both temporal and spatial locality are high. In this region, memory accesses are infrequent enough that queuing rarely occurs, so LZ4's faster decompression dominates. For example, at M7/B7, bandwidth inflation drops to $1.83\times$, and cache hit rate reaches 95%. Even in this most favorable region for LZ4, Zstd still provides 35% more effective capacity.

Temporal and spatial locality guide which configuration to use in latency-sensitive workloads. Higher spatial locality favors larger blocks, as denser within-page accesses amortize read amplification more effectively. Higher temporal locality favors larger blocks and faster algorithms, as fewer cache misses and less temporal clustering reduce queuing.

Access counts and burstiness modulate these trends, but the effect is secondary. Figure 17b shows the effect of working set size within each temporal locality level: higher access counts (larger working sets) favor larger blocks that can effectively reduce bandwidth inflation and prefer more complex algorithms, as the increased access intensity increases queuing. This mirrors the effect of higher spatial locality and lower temporal locality, producing the preference for larger blocks and Zstd.

Bandwidth inflation serves as a practical indicator of which algorithm to prefer. Figure 18 shows that below M5, LZ4's bandwidth inflation at 4KiB exceeds roughly 10, and in this regime Zstd wins on latency for the majority of traces. Architects can therefore use bandwidth inflation as a lightweight proxy: when it is higher than 10, the system is bandwidth-bound, and Zstd's better compression pays off.

IX. Conclusion

This paper is the first study of compression tradeoffs in CXL memory expanders at hyperscale. Read amplification and bandwidth inflation emerge as the primary metrics that determine expander performance. We discover that these behaviors cause more complex compression algorithms to simultaneously increase capacity and reduce latency. We also show that these findings do not transfer to widely used benchmarks, as they exhibit fundamentally different locality from production cold memory. To address this discrepancy, we develop *COLA*, the first parameterized locality model for cold memory with an integrated trace generator. Sweeping across a wide range of access patterns, we find that Zstd's advantage holds broadly, with LZ4 becoming competitive only under extremely high locality. Together, these results suggest that revisiting compression algorithms and block sizes can yield $> 30\%$ capacity savings while improving performance by 12%, making compression-based expanders a promising direction for future datacenters.

Acknowledgment

This work was supported in part by grants from PRISM and CogniSense, as well as by Semiconductor Research Corporation (SRC) programs sponsored by DARPA.

APPENDIX

A. Abstract

This artifact supports the main conclusions of the paper in four parts: (1) CXL memory-expander performance analysis using cold-memory traces with Google-fleet-like locality, (2) comparison of benchmark and Google-fleet-like cold-memory traces, (3) validation that COLA reproduces the locality and memory-expander performance of an input trace, and (4) a COLA parameter sweep showing when Zstd outperforms LZ4.

The original Google fleet traces cannot be released because they contain private and proprietary information. We instead release their reduced COLA parameters and use independently generated traces in the artifact. These traces preserve the locality properties relevant to cold-tier evaluation but are not copies of the internal traces. Consequently, experiments involving fleet data reproduce the paper's methodology, trends, and conclusions rather than its exact numerical results. For COLA representativeness, the artifact provides an end-to-end, publicly reproducible validation using benchmark cold-memory traces as the original inputs.

B. Artifact check-list (meta-information)

- **Algorithm:** COLA trace analysis and generation.
- **Program:** COLA, CXL-Tracer, analysis scripts, and a gem5-based CXL memory-expander simulator.
- **Binary:** gem5 simulator containing the evaluated device model.
- **Run-time environment:** Linux; exact dependencies are documented in the artifact repository.
- **Hardware:** A CXL-capable host is required only to recollect benchmark traces with CXL-Tracer.
- **Execution:** Command-line scripts organized by experiment.
- **Metrics:** Access latency, cache hit rate, bandwidth inflation, effective capacity, reuse-distance EMD, and footprint error.
- **Output:** Tables and plots corresponding to Tables 1–2 and Figures 5, 7–9, 11–13, and 15–18.
- **Experiments:** CXL expander analysis, benchmark/fleet-like comparison, COLA fidelity, and COLA sensitivity sweep.
- **How much disk space is required (approximately)?:** Less than 10GB if pre-provided datasets are used.
- **How much time is needed to prepare workflow?:** Less than an hour if pre-provided datasets are used.
- **How much time is needed to complete experiments?:** Less than 6 hours with 32 parallel simulations.
- **Publicly available?:** Partially. COLA, its parameters, generated traces, benchmark tracing support, and analysis scripts are public; raw Google traces are not.
- **Code licenses (if publicly available)?:** Apache 2.0 License
- **Workflow automation framework used?:** Shell/Python scripts.
- **Archived (provide DOI)?:** 10.5281/zenodo.22137537

C. Description

1) How to access: The COLA implementation, released parameter sets, trace generator, experiment scripts, and plotting scripts are provided on the `MICRO2026_artifact` branch of `https://github.com/google-systems-research/cold-memory-model`. The CXL-Tracer changes and benchmark-trace collection instructions are on the `MICRO_2026_COLA_artifact` branch of https://github.com/ece-fast-lab/cxl-tracer. The source code for the gem5 simulator is available at https://github.com/skyp0714/gem5_trace_cache, and we provide a prebuilt simulator in the artifact.

2) Hardware dependencies: Trace generation, simulation, analysis, and plotting require only a standard x86-64 Linux machine. Recollecting benchmark traces is optional and requires the CXL-Tracer hardware/software setup documented in its repository. The artifact also includes previously collected public benchmark traces so that reviewers without such a system can run the remaining workflow.

3) Software dependencies: The artifact uses the COLA repository, Python analysis packages, the supplied gem5 simulator binary, and standard compression libraries for LZ4 and Zstd. Exact versions and one-step environment setup are specified on the artifact branch. CXL-Tracer and its dependencies are needed only for optional trace recollection.

4) Data sets: The artifact contains (i) reduced COLA parameter sets derived from the ten Google fleet traces, (ii) synthetic Google-fleet-like traces generated from those parameters, and (iii) benchmark cold-memory traces collected using CXL-Tracer. Raw Google fleet traces are neither included nor recoverable from the released parameters or synthetic traces.

D. Installation

Clone the two artifact branches, install the pinned analysis dependencies, and place the simulator binary and benchmark traces at the location documented by the COLA repository. A smoke test generates a short trace, simulates it, and checks the output schema. CXL-Tracer installation is unnecessary when using the provided benchmark traces.

```
$ git clone
    https://github.com/google-systems-research/cold
    -memory-model.git
$ git switch MICRO2026_artifact
$ cd ./cold-memory-model
$ python3 -m venv .venv
$ ./.venv/bin/python -m pip install --upgrade pip
$ ./.venv/bin/pip install -r requirements.txt
$ ./.venv/bin/python -c \
  "from synthesis import parse_compressibility;
      parse_compressibility()"
```

E. Experiment workflow

1) E1: CXL expander performance on Google-fleet-like traces: Generate traces from the released Google-derived COLA parameters and run all compression algorithm, block-size, and cache-size configurations with the provided simulator. Aggregate access latency, effective capacity, tail latency, cache hit rate, and bandwidth inflation. This experiment corresponds to Table 1, Figure 5, and Figure 7. It is expected to reproduce the reported tradeoffs and trends, not paper-exact values, because independently generated traces replace the confidential originals.

```
# Table 1 + Figure 5
$ ./.venv/bin/python
    ae_reproduce_scripts/run_fig5_tab1.py
# Figure 7
$ ./.venv/bin/python
    ae_reproduce_scripts/run_fig7.py
```

2) E2: Benchmark versus Google-fleet-like traces: Use the supplied benchmark traces, or optionally recollect them with CXL-Tracer, and compare them with traces generated from the Google-derived COLA parameters. Report total accesses, accesses per page, and the traces' temporal and spatial locality coordinates. This experiment corresponds to Figures 8 and 9.

```
# Figures 8a, 8b, 9
$ ./ae_reproduce_scripts/run_fig8a_fig8b_fig9.sh
```

3) E3: COLA representativeness: Treat each public benchmark cold-memory trace as an original trace, extract its COLA parameters, generate an independent proxy trace, and compare the pair. The locality evaluation reports the reconstructed transition and bit-flip models, reuse-distance CDF and EMD, and per-page footprint. The expander evaluation compares cache hit rate and access latency across block sizes using the same simulator. This public substitute for the confidential original proxy comparison follows the methodology of Figures 13, 14, and 16.

```
# Figures 13, 14
$ ./.venv/bin/python
    ae_reproduce_scripts/run_fig13_fig14.py
# Figure 16
$ ./.venv/bin/python
    ae_reproduce_scripts/run_fig16.py
```

4) E4: COLA sensitivity study: Run the released parameter sweep over working-set size, temporal locality, spatial locality, access-count distribution, and burstiness. Simulate LZ4 and Zstd at the evaluated block sizes, identify the lowest-latency configuration, and compute bandwidth inflation. This experiment corresponds to Figures 17 and 18.

```
# Figures 17a, 17b, 18
$ ./.venv/bin/python
    ae_reproduce_scripts/run_fig17a_fig17b_fig18.py
```

F. Evaluation and expected results

E1 shows that memory bandwidth and queuing can make Zstd preferable to LZ4 despite Zstd's longer per-block decompression latency, and that block-cache benefits diminish beyond the relevant capacity range. E2 shows that benchmark and Google-fleet-like cold-memory traces occupy different locality regimes and can lead to different compression choices. E3 shows that a COLA-generated proxy preserves the original benchmark trace's locality and tracks its cache hit-rate and access-latency trends, including the preferred configuration. E4 shows that Zstd wins on both capacity and latency for most of the explored traces (approximately 75% in the paper's sweep), while LZ4 becomes competitive mainly at high temporal and spatial locality; high bandwidth inflation should favor Zstd.

Because COLA generation is stochastic and the confidential traces are replaced with synthetic or benchmark traces where necessary, small numerical variation is expected. The artifact scripts report both raw measurements and derived plots so reviewers can inspect the conclusions directly.

REFERENCES

[1] Open Compute Project, "2021 OCP global summit," https://www.opencompute.org/events/past-events/2021-ocp-global-summit, 2021, accessed: 2026-02-09.

[2] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill *et al.*, "Design tradeoffs in CXL-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.

[3] Reuters, "The AI frenzy is driving a memory chip supply crisis," https://www.reuters.com/world/china/ai-frenzy-is-driving-new-global-supply-chain-crisis-2025-12-03/, 2025.

[4] N. World, "Samsung warns of memory shortages driving industry-wide price surge in 2026," https://www.networkworld.com/article/4113772/samsung-warns-of-memory-shortages-driving-industry-wide-price-surge-in-2026.html, 2026.

[5] P. Chauhan, C. Petersen, B. Morris, and J. Glisse, "OCP hyperscale CXL tiered memory expander specification, revision 1 version 1.0," Open Compute Project Foundation, 2023, template v1.2 , effective 27 Oct 2023. [Online]. Available: https://www.opencompute.org/documents/hyperscale-tiered-memory-expander-specification-for-compute-express-link-cxl-1-pdf

[6] Marvell Technology, "Marvell Structera A 2504 CXL Near-Memory Accelerator," https://www.marvell.com/products/cxl.html, accessed: 2025-07-11.

[7] ZeroPoint Technologies, "DenseMem CXL memory expansion," https://www.zeropoint-tech.com/products/cxl-expansion-dense-memory, 2024, accessed: 2026-02-22.

[8] A. U. Haq, R. Xie, L. Ma, Y. Fang, L. Liu, and T. Zhang, "ZipCXL: CXL-based main memory compression at low performance penalty," in *Proceedings of the International Symposium on Memory Systems*, 2025, pp. 15–27.

[9] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 317–330.

[10] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 844–855, 2020.

[11] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 209–223.

[12] A. Van Renen, D. Horn, P. Pfeil, K. Vaidya, W. Dong, M. Narayanaswamy, Z. Liu, G. Saxena, A. Kipf, and T. Kraska, "Why TPC is not enough: An analysis of the amazon redshift fleet," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3694–3706, 2024.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[14] R. Shashidhara, S. Peter, S. Hare, and K. Keeton, "Closing the benchmark gap for tiered memory," in *Proceedings of the 3rd Workshop on Disruptive Memory Systems*, 2025, pp. 45–53.

[15] A. Awad and Y. Solihin, "Stm: Cloning the spatial and temporal memory access behavior," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 237–247.

[16] G. Balakrishnan and Y. Solihin, "West: Cloning data cache behavior using stochastic traces," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[17] R. K. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and accurate exploration of multi-level caches using hierarchical reuse distance," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 145–156.

[18] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: CXL-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[19] Y. Zhong, F. Kazhamiaka, P. Zardoshti, S. Teng, R. Fonseca, M. D. Hill, and D. S. Berger, "Octopus: Enhancing CXL memory pods via sparse topology," in *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, 2026, pp. 1303–1322.

[20] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," *SIGPLAN Not.*, vol. 52, no. 4, p. 631–644, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093336.3037706

[21] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, 2001.

[22] S. Karandikar, A. N. Udipi, J. Choi, J. Whangbo, J. Zhao, S. Kanev, E. Lim, J. Alakuijala, V. Madduri, Y. S. Shao *et al.*, "CDPU: Co-designing compression and decompression processing units for hyperscale systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–17.

[23] Xilinx, "Vitis libraries," https://github.com/Xilinx/Vitis_Libraries, 2024, accessed: 2025-07-11.

[24] J. Kang, Q. Xia, I. Jeong, Y. Park, and N. S. Kim, "Intel® in-memory analytics accelerator: Performance characterization and guidelines," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2025, pp. 1–13.

[25] Q. Xia, H. Ji, Y. Zhou, and N. S. Kim, "Hardware-accelerated kernel-space memory compression using Intel QAT," *IEEE Computer Architecture Letters*, vol. 24, no. 1, pp. 57–60, 2025.

[26] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic *et al.*, "Towards an adaptable systems architecture for memory tiering at warehouse-scale," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 727–741.

[27] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 495–513. [Online]. Available: https://doi.org/10.1145/3387514.3405883

[28] R. Xie, A. U. Haq, L. Ma, Y. Fang, Z. B. Engineer, L. Liu, and T. Zhang, "Reimagining memory access for LLM inference: Compression-aware memory controller design," 2025. [Online]. Available: https://arxiv.org/abs/2503.18869

[29] D. Dangwal, W. Cui, J. McMahan, and T. Sherwood, "Safer program behavior sharing through trace wringing," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1059–1072.

[30] Transaction Processing Performance Council, "TPC-C homepage," https://www.tpc.org/tpcc/, official benchmark description, accessed 2026-03-22.

[31] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 18–32.

[32] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, 2010, pp. 143–154.

[33] Redis, "Redis open source documentation," https://redis.io/docs/latest/get-started/, official documentation, accessed 2026-03-22.

[34] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[35] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *the Journal of machine Learning research*, vol. 9, pp. 1871–1874, 2008.

[36] Standard Performance Evaluation Corporation, "SPEC CPU® 2017 benchmark," https://www.spec.org/cpu2017/, 2017, accessed: 2026-03-22.

[37] R. van Riel and V. Chegu, "Automatic NUMA balancing," Red Hat Summit, 2014.

[38] S. Vanavasam, A. A. Cruz, and N. S. Kim, "CXL-Tracer: Accurate, full-system memory tracing on commercial hardware via CXL memory," *IEEE Computer Architecture Letters*, vol. 25, no. 1, pp. 114–117, 2026.

[39] H. Hadian, J. Liu, H. Xu, H. Idden, and H. Li, "PACT: A criticality-first design for tiered memory," in *ASPLOS '26: Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Pittsburgh, PA, USA: ACM, 2026.

[40] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive page migration policy with huge pages in tiered memory systems," *IEEE Transactions on Computers*, vol. 71, no. 1, pp. 53–68, 2020.

[41] D. Xu, J. Ryu, K. Shin, P. Su, and D. Li, "FlexMem: Adaptive page profiling and migration for tiered memory," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 817–833.

[42] K. Song, J. Yang, Z. Wang, J. Zhao, S. Liu, and G. Pekhimenko, "Hybridtier: an adaptive and lightweight CXL-memory tiering system," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2025, pp. 112–128.

[43] Y. Sun, J. Kim, Z. Yu, J. Zhang, S. Chai, M. J. Kim, H. Nam, J. Park, E. Na, Y. Yuan *et al.*, "M5: Mastering page migration and memory management for CXL-based tiered memory systems," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 604–621.

[44] M. Vuppalapati and R. Agarwal, "Tiered memory management: Access latency is the key!" in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 79–94.

[45] J. Liu, H. Hadian, H. Xu, and H. Li, "Tiered memory management beyond hotness," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 731–747.

[46] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "MIND: In-network memory management for disaggregated data centers," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, 2021, pp. 488–504.

[47] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 79–92.

[48] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 69–87.

[49] S. Li, Y. Zhou, H. Ren, and J. Huang, "ByteFS: system support for (CXL-based) memory-semantic solid-state drives," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 116–132.

[50] H. Zhang, Y. Xue, Y. E. Zhou, S. Li, and J. Huang, "SkyByte: architecting an efficient memory-semantic CXL-based SSD with OS and hardware co-design," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 577–593.

[51] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 2, pp. 184–215, 1989.

[52] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 245–257.

[53] W. S. Wong and R. J. T. Morris, "Benchmark synthesis using the LRU cache hit function," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 637–645, 1988.

[54] R. Panda and L. K. John, "HALO: A hierarchical memory access locality modeling technique for memory system explorations," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 118–128.

[55] E. Strohmaier and H. Shan, "Apex-Map: A global data access benchmark to analyze HPC systems and parallel programming paradigms," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005, pp. 49–49.