ZipCXL: CXL-based Main Memory Compression at Low Performance Penalty

Asad Ul Haq
Rensselaer Polytechnic Institute
Troy, NY, USA
asadul@rpi.edu

Rui Xie
Rensselaer Polytechnic Institute
Troy, NY, USA
xier2@rpi.edu

Linsen Ma
Rensselaer Polytechnic Institute
Troy, NY, USA
mal3@rpi.edu

Yunha Fang
Rensselaer Polytechnic Institute
Troy, NY, USA
fangy6@rpi.edu

Liu Liu
Rensselaer Polytechnic Institute
Troy, NY, USA
liu.liu@rpi.edu

Tong Zhang
Rensselaer Polytechnic Institute
Troy, NY, USA
zhangt4@rpi.edu

Abstract

The escalating cost of DRAM and the typically high compressibility of memory content make main memory compression highly desirable. However, its practical deployment has been hindered by significant challenges, including its adverse impact on performance and, more critically, the substantial integration challenges it poses to computing infrastructure. The emerging Compute Express Link (CXL) ecosystem provides a unique opportunity to implement main memory compression with minimal integration overhead, shifting the primary adoption barrier towards performance impact. This paper tackles this challenge by introducing three simple yet effective design techniques to enhance the design of compression-capable CXL memory controllers. The first two techniques improve the trade-off between compression ratio and speed performance by dynamically adjusting compression configurations in adaptation to runtime data characteristics. The third technique mitigates compression-induced speed performance degradation by decoupling the in-memory placement of compressed data blocks from their associated error correction code (ECC) redundancy. To evaluate these techniques, we performed RTL-level design and synthesis to estimate silicon cost overhead and developed a simulation platform to capture the trade-offs between compression ratio and speed performance. The results demonstrate that the proposed techniques effectively improve compression ratio vs. performance trade-offs with negligible silicon cost overhead.

CCS Concepts

• **Computer systems organization** → *Data flow architectures*; • **Hardware** → **Very large scale integration design**.

Keywords

Main memory compression, CXL, Adaptive compression, ECC, VLSI architecture

1 Introduction

The high cost of DRAM has become a critical concern in modern computing systems, with memory expenses accounting for a significant portion of the total cost of ownership (TCO). For instance, Microsoft Azure recently reported [19] that DRAM already accounts for more than 50% of their server costs. Given the typically high compressibility of memory-resident content, main memory compression appears to be a promising solution to this memory cost crisis. However, despite decades of research efforts [2, 10, 18, 29, 33, 38], main memory compression has struggled to transition from academic exploration to real-world deployment. The two primary obstacles are the performance degradation and the complexity of integrating memory compression into existing computing infrastructure.

The advent of Compute Express Link (CXL) offers an unprecedented opportunity to overcome the system integration challenges. By decoupling memory resources from host processors, CXL enables the realization of main memory compression with minimal disruption to existing infrastructure. This has generated significant industry interest. For instance, in 2023 the Open Compute Project (OCP) released a specification for compression-capable CXL memory devices, co-authored by Google and Meta [24], and in 2024 Marvell announced the world's first compression-capable CXL memory controller chip [15]. Unlike prior research that primarily focused on per-cacheline compression, intra-CXL compression targets large block sizes (e.g., 1KB and 4KB as specified by OCP). This approach generally achieves much higher compression ratios at the cost of longer data access latency. Such trade-offs can be acceptable because CXL memory devices are expected to serve as warm/cold memory tiers beneath DRAM DIMMs directly connected to host processors. Memory tiering is well justified by the significant memory access intensity variation, e.g., Microsoft Azure reported [19] that ~50% of their VMs touch less than half of their rented memory.

Nonetheless, reducing the performance impact of compression remains highly desirable and determines how widely compression-capable CXL memory devices will be adopted. Compression affects the speed performance of CXL memory devices in three ways. First, a mapping table should be used to track the location and size of each compressed data block. Before reading or writing a data block, CXL memory controller must access this table, introducing latency overhead. Second, the mismatch between the host access granularity (e.g., 64B cacheline) and the compressed block sizes (e.g., 1KB or 4KB) amplifies DRAM bandwidth usage. Fetching even a single cacheline requires reading and decompressing the entire block. Third, despite the use of customized high-speed (de)compression hardware engines in CXL memory controller, the decompression latency during reads can still impose notable performance overhead.

This paper focuses on addressing the latter two challenges: bandwidth amplification and decompression overheads. The first challenge, involving mapping table access latency, can be mitigated by integrating an on-chip mapping table SRAM cache on CXL memory controllers, which can leverage extensive prior research on cache design. To tackle compression-induced bandwidth amplification and decompression overhead, this paper introduces three simple yet effective design techniques. The first two techniques empower CXL memory controllers to achieve improved trade-offs between compression ratio and speed performance, motivated by two key observations. First, while larger compression block sizes generally improve compression ratios, the specific correlation between block size and compression ratio varies across data blocks. Second, entropy coding (e.g., Huffman coding [14]) can augment LZ search [39] to further improve compression ratios (e.g., zlib [12] and ZSTD [5]). However, the compression ratio gain brought by entropy coding can vary significantly between data blocks. Furthermore, entropy decoding, due to its bit-serial processing nature, suffers from low throughput. Accordingly, this paper proposes *adaptive compression granularity* and *adaptive entropy coding bypassing*. These two techniques share the theme of adaptively configuring compression settings (i.e., compression block size and entropy coding ON/OFF) to improve trade-offs between compression ratio and speed performance. Moreover, since straightforward implementation of adaptive compression granularity is subject to significant silicon cost overhead, we further propose a *fused dual-stream* compressor VLSI architecture to reduce the silicon cost.

The third technique addresses DRAM bandwidth amplification by enhancing the bandwidth utilization efficiency. To ensure data integrity, CXL memory devices utilize ECC (error correction code) and deploy ECC DIMMs (e.g., 8+2 DDR5 DIMMs). In conventional practice, each ECC codeword is written to and fetched from ECC DIMM altogether. The ECC-induced DRAM bandwidth usage overhead is non-negligible (e.g., 20% in the case of 8+2 DDR5 DIMM). This is the price one must pay to minimize random cacheline access latency. Compression-capable CXL memory devices, however, operate on compressed blocks significantly larger than cachelines. Moreover, due to very low raw error rates of DRAM devices, error correction is rarely needed during read operations. This motivates the *data and ECC redundancy disaggregation* technique, which avoids fetching ECC redundancy from DRAM unless necessary. By doing so, CXL memory devices can use nearly 100% of their internal DRAM bandwidth for transferring user data.

We carried out experiments and simulations to evaluate the proposed design techniques and show the trade-offs. Using Synopsys Design Compiler, we conducted RTL-level design and synthesis to demonstrate the effectiveness of the proposed fused dual-stream compressor VLSI architecture. Leveraging the popular DRAM simulator DRAMSim3 [20],we built a simulation platform in support of compression-capable CXL memory devices, and carried out simulations under a variety of synthetic and real workloads. The simulation results quantitatively show that the proposed techniques can notably improve the compression ratio vs. speed performance. The rest of the paper is organized as follows: Section 2 reviews the background and discusses the realization of compression-capable CXL memory devices. Section 3 presents the three proposed techniques for mitigating compression-induced performance degradation. Section 4 and Section 5 discuss the evaluation methodology and present experimental results. Finally, Section 6 summarizes related prior work and Section 7 draws the conclusion.

2 Background

2.1 Main Memory Compression

Although main memory compression has been well researched [2, 10, 18, 29, 33, 38], it has struggled to gain real-world adoption due to unfavorable benefit vs. cost trade-offs. To minimize performance impact and simplify system integration, prior research primarily focused on per-cacheline compression [2, 17, 29]. While this approach helps to lessen system integration challenge and system speed performance impact, it delivers much lower compression ratios compared to coarse-grained, general-purpose compression algorithms. Before the advent of CXL [8], host processors were solely responsible for implementing main memory compression, requiring complex architectural modifications. This added significant complexity to already intricate CPU designs and substantially increased system integration difficulty. To realize the cost-saving potential of memory compression, additional mapping table is essential to handle the unpredictable, variable lengths of compressed data. Despite the use of hardware-accelerated (de)compression, main memory compression faces notable latency overhead from mapping table access, particularly in the case of fine-grained per-cacheline compression. Furthermore, with fixed burst length of DRAM devices, DRAM DIMMs are optimized for fixed-size cacheline accesses. Variable-length compressed cachelines often misalign with this fixed-size structure, further exacerbating latency. Collectively, these factors cause the unfavorable benefit-to-cost trade-offs that have limited the adoption of main memory compression in real-world systems.

2.2 CXL: Revive Main Memory Compression

Built upon the mature PCIe ecosystem, CXL has been supported by modern server CPUs, enabling coherent data transfer between CPUs and attached devices [8]. Among various types of CXL devices, CXL memory devices have attracted most attentions [23, 31, 35], with all the major DRAM manufacturers already announcing CXL memory products. One of the advanced features proposed for these devices is main memory compression, a capability that aligns with CXL's goal of providing a cost-effective and scalable memory tier.

In contrast to prior research, this time the focus is on coarse-grained compression, targeting significant cost savings. In 2023 OCP released a specification for CXL memory devices that support 1KB~4KB compression [24], and in 2024 Marvell announced the world's first compression-capable CXL memory controller chip [15] compliant to the OCP specification. While coarse-grained compression incurs DRAM read/write amplification and performance degradation, this trade-off can be acceptable since compression-capable CXL memory devices are intended to function as warm or cold memory tiers. Despite using larger-than-cacheline compression block size, such CXL memory devices still can leverage prior research [11, 21, 28] to manage the storage of variable-length compressed data. Memory compression inherently involves a trade-off

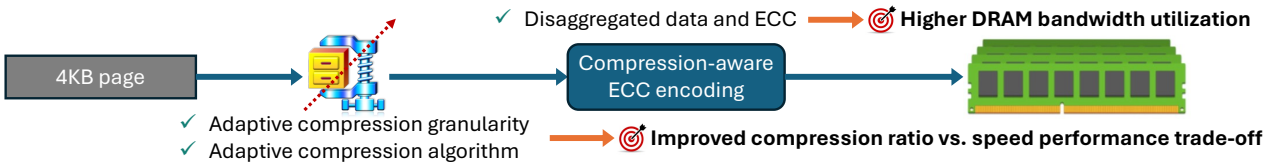

Figure 1: Overview of the proposed three design techniques as the cornerstone of the ZipCXL solution.

between compression ratio and performance. Regarding the implementation of (de)compression hardware data path, one may adopt the following general principles:

• *Matching DRAM bandwidth with multiple hardware accelerators*: Let n_{ch} denote the number of DDR channels inside CXL memory devices and B_{ch} represent the per-channel bandwidth. The total (de)compression throughput should match the aggregated bandwidth $n_{ch} \cdot B_{ch}$. For DDR4 and DDR5, the per-channel bandwidth B_{ch} can easily exceed 25GB/s. With just four DDR channels, the aggregated bandwidth can surpass 100GB/s, far beyond the maximum achievable throughput of a single hardware (de)compression accelerator. Hence, CXL memory controllers must integrate multiple hardware accelerators to match the aggregated DRAM bandwidth.

• *Silicon-cost-driven compression*: CXL memory controllers should integrate an on-chip SRAM write buffer to exploit memory write locality while hiding compression latency from host CPU. Due to the high cost of implementing LZ search [1], we should minimize the total silicon area of compression accelerators. Let n_{com} denote the number of compression accelerators. To match the total DRAM bandwidth, each compression accelerator must achieve a throughput of $B_{com} = \frac{n_{ch} \cdot B_{ch}}{n_{com}}$. Let $f_A(B_{com})$ represent the silicon area required for a single throughput-B_{com} accelerator. The on-chip write buffer, which hides compression latency from the host, provides flexibility to optimize the trade-off between the number of accelerators (n_{com}) and per-accelerator throughput (B_{com}). This optimization minimizes the total silicon cost of compression accelerators by solving:

$$\min_{\forall n_{com}} \left(n_{com} \cdot f_A\left(\frac{n_{ch} \cdot B_{ch}}{n_{com}}\right)\right). \tag{1}$$

• *Latency-driven decompression*: Decompression lies on the critical path of memory reads, directly impacting system performance. Due to the sequential nature of LZ decompression, one compressed block can only be decompressed by a single decompressor. Nonetheless, this sequential nature also enables *pipelining*, allowing DRAM-to-controller data transfer and decompression to overlap: As each byte is transferred from DRAM to CXL memory controller, it is immediately fed into the decompression accelerator. Let B_{dec} denote the throughput of a decompression accelerator, $m_{st} \geq 1$ represent the number of DDR channels over which each compressed data block stripes, and τ_{DRAM} denote the internal DRAM read latency (e.g., including the delay of row activation delay, column access strobe, and row precharge). For a size-s_c compressed block, the pipelined read path has a latency of

$$\tau_{DRAM} + \frac{s_c}{\min\left(B_{dec}, m_{st} \cdot B_{ch}\right)}. \tag{2}$$

Since per-channel bandwidth B_{ch} can exceed 25GB/s and it is challenging for B_{dec} to surpass 50GB/s, we should set $m_{st} = 1$.Hence, if $B_{dec} > B_{ch}$, decompression operation on its own does not incur notable additional latency. The achievable throughput of a single decompression accelerator B_{dec} depends on the compression algorithm. For example, entropy coding relies on bit-serial operations, which limits hardware implementation parallelism. Consequently, achieving $B_{dec} > B_{ch}$ is much more feasible with LZ-only compression (e.g., LZ4) than with LZ+entropy compression (e.g., zlib).

3 Proposed Design Techniques

This section presents three design techniques, as illustrated in Fig. 1, to enhance (de)compression hardware data path inside CXL memory controllers. The first two improve trade-offs between compression ratio and speed performance, while the third one reduces performance impact by increasing effective DRAM bandwidth utilization.

3.1 Adaptive Compression Granularity

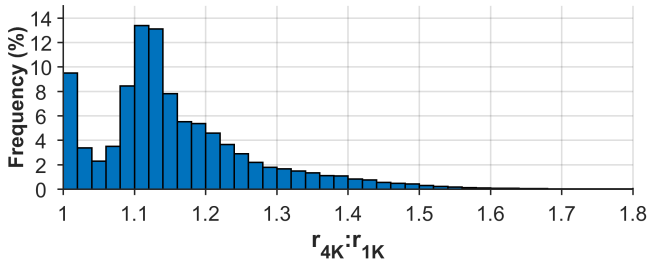

Figure 2: Measured histogram of $r_{4K} : r_{1K}$, the relative compression ratio improvement of 4KB vs. 1KB compression.

The choice of compression block size (e.g., 1KB vs. 4KB) significantly influences the trade-off between compression ratio and speed performance. Larger block sizes generally achieve better compression ratios by offering a wider window for redundancy elimination. However, the correlation between block size and compression ratio varies across data blocks. To demonstrate this variability, we conducted experiments on 4KB blocks randomly extracted from several open data repositories [3, 9, 16, 30, 37]. For each 4KB data block, let $r_{4K} \geq 1$ denote the compression ratio[1] when compressed as a single block, and $r_{1K} \geq 1$ represent the compression ratio when split into four 1KB blocks and compressed individually. Fig. 2 shows the $r_{4K} : r_{1K}$ ratio histogram. The results reveal a wide distribution

[1]In this work, the compression ratio is defined as the original data block size divided by the compressed block size and is therefore always ≥ 1.

of $r_{4K} : r_{1K}$, confirming that the correlation between block size and compression ratio is highly content-dependent.

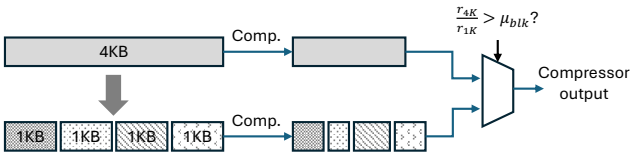

Figure 3: Adaptive compression block size selection.

Motivated by this observation, we propose a technique called *adaptive compression granularity* that dynamically configures the compression block size based on its correlation with compression ratio. Let l_{blk} denote the compression block size. For this study, we assume the CXL memory controller supports two configurations: l_{blk} =4KB and l_{blk} =1KB, as defined by the OCP specification. For each 4KB memory page, CXL memory controller has two options: (1) compresses entire 4KB memory page as a single block (l_{blk} =4KB), while (2) partitions the 4KB memory page into four 1KB blocks and compresses each block individually (l_{blk} =1KB). To simplify data management, we assume that regardless of the compression block size, each compressed 4KB memory page is stored contiguously in DRAM and associates with a single entry in the mapping table. This assumption ensures that the choice of block size does not add unnecessary complexity to the mapping mechanism. As illustrated in Fig. 3, we use the *compression advantage factor* $\frac{r_{4K}}{r_{1K}} \geq 1$ to quantify the relative compression ratio improvement from using l_{blk} =4KB over l_{blk} =1KB. We compare this compression advantage factor with a predefined threshold μ_{blk} to decide the generation of final compression output. This enables the use of l_{blk} =1KB only for data whose compressibility is lest sensitive to the compression block size. Therefore, we can favorably trace the trade-off between compression ratio and speed performance by adjusting the threshold μ_{blk}.

Despite its conceptual simplicity, implementing an efficient dual-window LZ compression accelerator poses a significant silicon cost challenge. A straightforward approach would require two separate compression engines, one for l_{blk} = 4KB and another for l_{blk} = 1KB, leading to a substantial increase in hardware overhead. Given that compression is already a resource-intensive task, such duplication would make implementation prohibitively expensive. To address this issue, we propose a fused dual-stream LZ compressor that leverages the inherent operational similarity between LZ searches with different window sizes to enable resource sharing. Our design builds upon the LZ compression architecture developed by IBM for its z15 CPU [1], which stores the search window using a hybrid near/far content-addressable memory (CAM) fabric. Motivated by the observation that most LZ search matches occur within a short distance, this design can achieve an improved balance between LZ search quality (hence compression ratio) and silicon cost. Interested readers are referred to [1] for further details.

Figure 4(a) illustrates a conventional adaptive compression block chunking implementation using two separate LZ compressors. The near CAM, implemented with shift registers, retains the most recent data and enables exhaustive pattern matching for maximum LZ search accuracy. Meanwhile, the far CAM, implemented as an SRAM-based hash table, provides an approximate yet cost-effective content-addressable search at a modest accuracy loss. In contrast, Figure 4(b) presents our fused dual-stream LZ compressor, which performs 1KB and 4KB LZ searches simultaneously while maintaining a single shared search window content. Instead of duplicating search window content across two separate compressors, our approach retains one copy within a unified near CAM and far CAM structure. To support simultaneous LZ searches across both window sizes, these CAMs are equipped with dual-port access and dual-port pattern matching capabilities. Additionally, the 1KB window constraint is enforced at the compression output stage to ensure correctness. Since CAMs dominate the silicon area of LZ compressors, our fused design reduces the hardware cost by over 20% compared to the straightforward dual-compressor implementation. To validate our approach, we implemented an RTL design and performed synthesis at the 45nm node. The results confirm significant area savings and preserved compression efficiency, demonstrating the effectiveness of our design. Further details will be provided in Section 5.

3.2 Adaptive Entropy Coding Bypassing

Beyond compression block size, the decision on the usage of entropy coding also significantly influences the trade-off between compression ratio and speed performance. While appending entropy coding to LZ compression typically improves compression ratios, the actual compression ratio gains can vary greatly across data blocks. This work focuses on implementing entropy coding in the form of Huffman coding. To demonstrate this variability, we conducted experiments on representative datasets [3, 9, 16, 30, 37]. Let r_{LZ} denote the compression ratio achieved with LZ compression alone and $r_{LZ+Huff}$ represent the compression ratio when Huffman coding is appended to LZ. Fig. 5 shows the histogram of $r_{LZ+Huff} : r_{LZ}$ ratio, revealing substantial variation in the effectiveness of Huffman coding on further improving compression ratio. Meanwhile, Huffman decoding heavily involves bit-level operations, which inherently limit its achievable throughput under reasonable silicon cost. Realizing a decoding throughput that matches DDR channel bandwidth (e.g., 25GB/s and above) is extremely challenging, if not impossible. As a result, the Huffman decoder will almost surely be the throughput bottleneck on the memory read data path. Nevertheless, appending Huffman coding does not always increase memory read latency. Let B_{Huff} denote the Huffman decoding throughput, and assume B_{ch}, the DDR channel bandwidth, is lower than the LZ decompression throughput. According to Eq. 1, the extra memory read latency introduced by Huffman coding can be expressed as

$$\begin{aligned} \tau_{ext} &= \frac{l_{blk}}{r_{LZ+Huff}} \cdot \frac{1}{B_{Huff}} - \frac{l_{blk}}{r_{LZ}} \cdot \frac{1}{B_{ch}} \\ &= \frac{l_{blk}}{r_{LZ} \cdot B_{ch}} \cdot \left(\frac{r_{LZ} \cdot B_{ch}}{r_{LZ+Huff} \cdot B_{Huff}} - 1 \right). \end{aligned} \tag{3}$$

Appending Huffman coding reduces read latency ($\tau_{ext} < 0$) when $r_{LZ} \cdot B_{ch} < r_{LZ+Huff} \cdot B_{Huff}$, as the improved compression ratio sufficiently offsets the decoding bottleneck. This suggests a dynamic approach to *bypass* Huffman coding based on its runtime

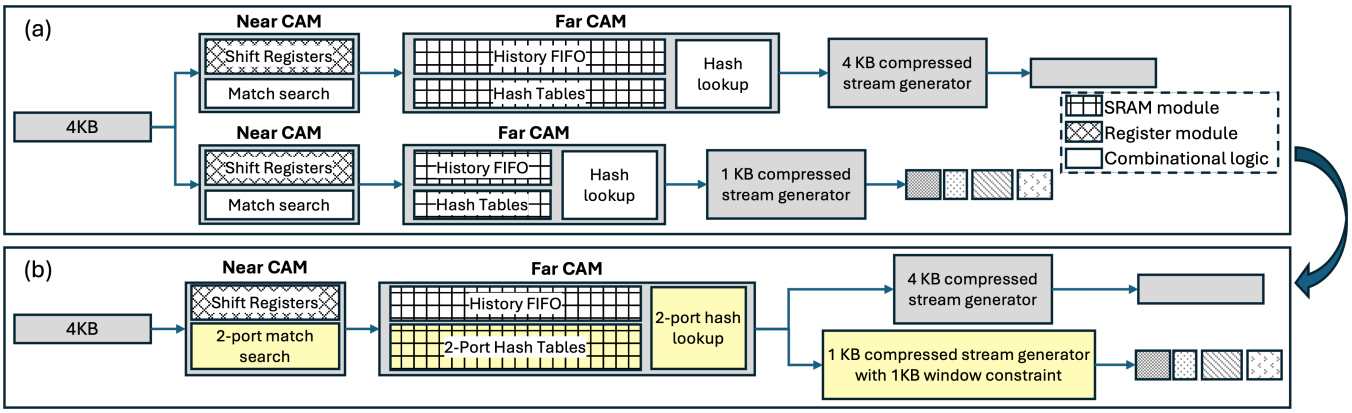

Figure 4: Realize adaptive compression block chunking using (a) two separate LZ compressors and (b) a low-cost fused dual-stream LZ compressor.

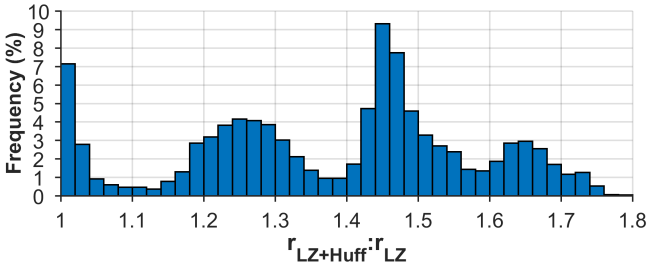

Figure 5: Histogram of $r_{LZ+Huff} : r_{LZ}$, the relative compression ratio improvement by switching to $r_{LZ+Huff}$ from r_{LZ}.

effectiveness. We define $\varphi = B_{Huff}/B_{ch} < 1$ and introduce a design parameter $\mu_{Huff} \geq 1$, which allows the CXL memory controller to configure the final compression output as illustrated in Fig. 6:

- *Use the output of LZ + Huffman compression*: If we have $r_{LZ+Huff} \geq \frac{r_{LZ}}{\varphi \cdot \mu_{Huff}}$, the compression ratio gain enabled by Huffman coding is sufficient to outweigh its decoding-induced read latency overhead. In this case, the output of LZ compression plus Huffman coding is used.
- *Use the output of LZ compression*: If $r_{LZ+Huff} < \frac{r_{LZ}}{\varphi \cdot \mu_{Huff}}$, the compression ratio gain from Huffman coding does not sufficiently offset its decoding-induced read latency overhead. As a result, only the LZ compression output is used.

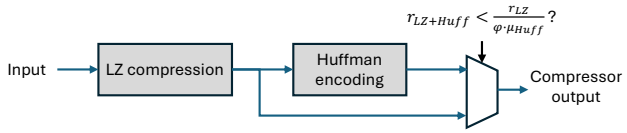

Figure 6: Illustration of adaptive entropy coding bypassing.

This enables the use of Huffman coding only for data that can benefit the most from Huffman coding. Therefore, we can favorably trace the optimal trade-off between compression ratio and speed performance by adjusting the threshold μ_{Huff}. Compared to always using LZ compression alone, this approach introduces additional silicon cost due to the need for Huffman codecs. However, silicon cost of a Huffman encoder is significantly lower than that of an LZ compressor, as it requires much less SRAM [32]. In contrast, due to intensive bit-level operations in Huffman decoding, achieving high Huffman decoding throughput demands aggressive use of techniques such as look-ahead decoding [26, 27], which can be very costly. In practice, the CXL memory controller should implement Huffman decoders with the highest feasible parallelism within the given silicon cost constraints. This ensures that Huffman coding can be effectively leveraged when its benefits justify the additional latency overhead.

3.3 Data-ECC Disaggregation

To ensure reliability, CXL memory devices employ ECC and utilize ECC DIMMs. Each rank of an ECC DIMM consists of $n_u + n_e$ DRAM devices, where n_u and n_e denote the number of user data devices and ECC devices, respectively. In DDR5 ECC DIMMs, n_u =8 and n_e is either 1 or 2, corresponding to different trade-off between memory cost and reliability. In conventional practice, a single ECC codeword protects one 64-byte cacheline with $m = 64 \cdot n_e/n_u$ bytes of coding redundancy. Each ECC codeword spans $n_u + n_e$ DRAM devices at the same address to minimize access latency. As a result, during read operations, only $\frac{n_u}{n_u+n_e}$ of the total DRAM bandwidth is used for transferring user data, while the remaining $\frac{n_e}{n_u+n_e}$ of bandwidth is consumed by ECC redundancy. Such ECC-induced bandwidth usage can be significant, e.g., 20% in 8+2 DDR5 ECC DIMMs.

Evidently, if we could reduce the ECC-induced bandwidth usage, the effective DRAM bandwidth for user data will increase, directly enhancing the speed performance of compression-capable CXL memory devices. An intuitive approach is to reduce the proportion of ECC redundancy by increasing the ECC codeword length. It is motivated by the well-known principle that longer ECC codewords can achieve the same error correction strength with less redundancy. Specifically, instead of protecting each 64-byte cacheline individually as in current practice, the entire compressed block could be protected by a single, long ECC codeword. While this

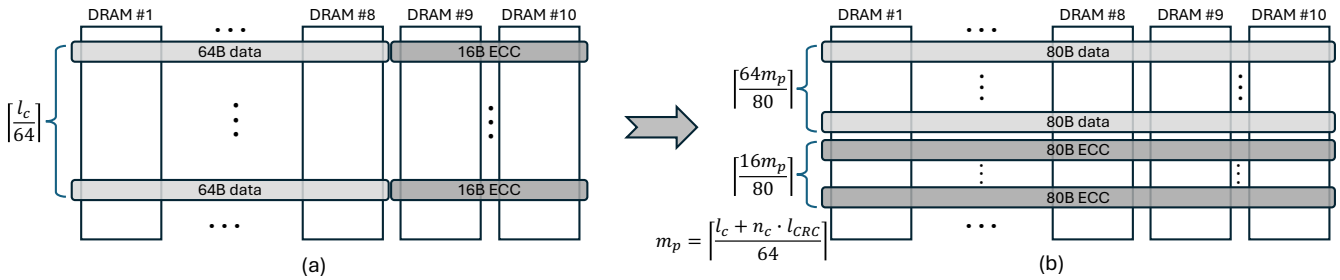

Figure 7: Illustration of data and ECC redundancy placement in 8+2 DDR5 DIMMs when using (a) conventional practice, and (b) proposed disaggregated design approach.

approach seems promising, it faces two problems: (1) Longer ECC codewords significantly increase the ECC encoder/decoder silicon implementation cost. (2) The error correction strength advantage of longer codeword length will diminish or completely disappear in the presence of long burst errors or catastrophic device failures.

We propose a technique to reduce ECC-induced bandwidth usage while maintaining the same amount of ECC redundancy (hence the same data reliability) as conventional practice. The technique leverages a key insight: under normal operating conditions, the raw error rate of modern DRAM is so low[40] that error correction is rarely invoked by memory controllers. This reliability is demonstrated by the widespread use of non-ECC DIMMs in consumer-grade computing systems. Therefore, in most cases, ECC serves only as an error detection mechanism to verify data integrity, while its error correction capability remains unused. Building on this observation, we propose to decouple the storage and retrieval of compressed data blocks and their associated ECC redundancy. Additionally, we append a lightweight CRC (Cyclic Redundancy Check) to each compressed data block to serve as the primary error detection mechanism. During a read operation, only the data block and its CRC are retrieved from DRAM. If the CRC check passes, as it does in the vast majority of cases due to DRAM's inherent reliability, the operation completes without fetching ECC redundancy. If the CRC check fails, the ECC redundancy is fetched, and error correction is performed to reconstruct the compressed data block. This can reduce the ECC-induced bandwidth usage to almost zero, allowing DRAM bandwidth to be nearly fully dedicated to transferring user data during normal read operations. Moreover, to enhance error detection effectiveness, we could partition one compressed block into multiple segments, each segment being protected by one CRC.

Fig. 7 illustrates the implementation of this scheme for 8+2 ×4 DDR5 DIMMs. In ×4 DDR5 DRAM, the burst length is 16, making the basic access unit a 80-byte stripe across the 10 DRAM devices in 8+2 DIMMs. Each ECC codeword protects 64 bytes of user data with 16 bytes of ECC redundancy. For a compressed data block of size l_c, the conventional approach requires $m_c = \lceil l_c/64 \rceil$ 80-byte ECC codewords to provide protection, with both the data and ECC redundancy of each ECC codeword stored together across the same address in DRAM. In the proposed design, n_c l_{CRC}-byte CRCs are embedded into the compressed data block, forming an expanded data block of size $(l_c + n_c \cdot l_{CRC})$ bytes. This expanded data block is protected by $m_p = \lceil (l_c + n_c \cdot l_{CRC})/64 \rceil$ 80-byte ECC codewords. Following the principle of data and ECC redundancy disaggregation, the $(64 \cdot m_p)$-byte user data is stored contiguously across all 10 DRAM devices, spanning $\lceil 64 \cdot m_p/80 \rceil$ 80-byte stripes. Similarly, the $(16 \cdot m_p)$-byte ECC redundancy is stored contiguously, spanning $\lceil 16 \cdot m_p/80 \rceil$ 80-byte stripes. The additional storage, in terms of the number of 80-byte stripes, required by the proposed design is given by

$$N_{ext} = \left\lceil \frac{64 \cdot m_p}{80} \right\rceil + \left\lceil \frac{16 \cdot m_p}{80} \right\rceil - m_c. \tag{4}$$

Since n_c can be only 2 or 4 and l_{CRC} is relatively very small (e.g., 16-byte) compared to the size of a typical compressed block, m_p is equal to either m_c or $m_c + 1$. As a result, N_{ext} is typically 0 or 1, indicating that the proposed design induces negligible storage space overhead in DRAM. Finally, let us quantitatively exam the CRC check failure probability. Let p_e denote the DRAM bit error rate, the CRC check failure probability can be expressed as

$$P_{CRC} = 1 - (1 - p_e)^{\left\lceil \frac{64 \cdot m_p}{80} \right\rceil \cdot 640}. \tag{5}$$

Accordingly to a recent study [40], DRAM exhibits bit error rates on the order of 10^{-11}. Even assuming DRAM bit error rate as high as 1×10^{-9}, we have that CRC check failure probability is well below 0.01% when l_c is 2KB and l_{CRC} is 16B and n_c is 4. By separating data and ECC redundancy and leveraging DRAM's high raw reliability, this proposed technique notably improves DRAM bandwidth utilization with minimal storage space overhead while maintaining the same data integrity.

4 Evaluation Methodology

To evaluate our proposed design techniques, we developed a CXL memory simulation platform based on DRAMsim3 [20]. This platform features a compression-capable controller and an underlying DRAM subsystem. The controller handles key functions, including mapping table maintenance, write buffering and caching, compression and decompression, request scheduling, and efficient storage management of compressed data blocks in DRAM. To simplify memory management, we employ slab allocation for compressed data blocks, eliminating the need for background garbage collection. The simulator supports all three proposed design techniques with either 1KB or 4KB compression granularity. Regardless of the chosen granularity, each mapping table entry corresponds to a 4KB memory page. When a 4KB memory page is compressed using a

1KB block size, all four compressed 1KB blocks are stored contiguously in DRAM, and the mapping table entry records the size of each compressed 1KB chunk within the 4KB page.

When a read request arrives from the host, the controller first checks its mapping table cache. A cache miss triggers a fetch from the DRAM-resident mapping table. The request is then queued, translated into DRAM transactions, and dispatched to the DRAM subsystem. On the read data path, data transfer from DRAM to the controller and subsequent decompression are fully pipelined to minimize or eliminate decompression-induced read latency overhead. As discussed in Section 2.2, each compressed block is stored within a single DDR channel, which is paired with a dedicated decompressor. The throughput of an LZ decompressor consistently exceeds the bandwidth of a DDR channel. However, due to the serial nature of Huffman decoding, its throughput is lower than the DDR channel bandwidth. For write requests, the controller first buffers data in on-chip SRAM and immediately acknowledges completion to the host, reducing host-perceived write latency. In the background, data evictions from the write buffer to DRAM occur through read-modify-write operations, with corresponding updates to the mapping table entries. The simulator assumes an implementation with a sufficient number of compressors to match the aggregated DRAM bandwidth.

When processing each host read request, our simulator records a detailed latency breakdown, including mapping table access latency, queuing delays, DRAM operations (such as row activation, closure, and data transfer), and any additional decompression-induced latency. The latencies of the DRAM DDR controller IP and the CXL/PCIe interface IP are modeled as 27ns and 40ns, respectively. To ensure realistic compression workload characterization, we employ diverse datasets from sources such as the Silesia corpus [9], Amazon AWS [3], Quandl [30], Kaggle [16], and TPC-H memory dumps [37]. Our simulation platform is highly configurable, enabling a comprehensive range of experimental setups. The granularity of adaptive LZ compression and adaptive Huffman encoding are controlled by parameters μ_{blk} and μ_{Huff}, respectively. Additional configurable settings include host queue depth, read/write ratios, data and ECC disaggregation, Huffman encoding/decoding throughput, and various DRAM subsystem parameters. For the experiments presented in this paper, we evaluated our design using standard DDR4 and DDR5 configurations, as detailed in Table 1.

DDR4 organization	4 channels, 3200 MT/s
DDR5 organization	4 channels, 6400 MT/s
DDR 4 timing	tCK: 0.625 ns, tCL: 22, tRCD: 22, tRP: 22
DDR 5 timing	tCK: 0.312 ns, tCL: 52, tRCD: 52, tRP: 52

Table 1: DRAM sub-system parameters in simulations.

5 Evaluation Results

This section presents a series of experiments conducted using the simulation platform and datasets described in Section 4. We begin by establishing a baseline with static compression configurations and conventional ECC data placement. Next, we evaluate each proposed design technique individually and in combination. The results demonstrate that our design solutions effectively improve the trade-off between compression ratio and performance compared to the baseline. Finally, we present and analyze the RTL-level implementation results of the fused dual-stream LZ compressor.

5.1 Baseline Evaluation

We first evaluate the baseline scenario using static compression configurations with fixed compression algorithm and block size and conventional ECC codeword placement. For comparison, we also include a scenario without compression, where each 64-byte cacheline can be directly accessed without any read/write amplification. Fig. 8(a) presents the average latency of random 64-byte reads under different compression configurations and queue depths. Since random 64-byte reads prevent effective use of the on-chip mapping table cache, each read request requires the controller to first fetch the mapping entry from DRAM and then retrieve and decompress the corresponding compressed data block. For LZ+Huffman compression, we configure each Huffman decoder with a throughput of 10GB/s, approximately 20% of a single DDR5 channel's bandwidth.

The results show a read latency of approximately 150ns (at a queue depth of 1) in the absence of compression, aligning with reported measurements from first-generation hardware CXL memory devices [22]. Under the same queue depth and with DDR4 memory, the read latency for LZ-only compression is 217ns, 235ns, and 270ns for block sizes of 1KB, 2KB, and 4KB, respectively. The relatively small latency variation across different block sizes is due to the fact that block data DRAM transfer time accounts for only 20% of the total end-to-end latency on average. Notably, the OCP specification targets a 250ns read latency for a 4KB compression block size. In contrast, LZ+Huffman compression results in a significantly higher read latency of approximately 332ns, primarily due to the lower throughput of Huffman decoding. As the queue depth increases, read latency also rises due to queuing effects. Additionally, with LZ-only compression, the latency difference between DDR4 and DDR5 becomes more evident at higher queue depths, as the impact of data transfer latency is amplified by queuing. In contrast, when using LZ+Huffman compression, the read latencies for DDR4 and DDR5 remain similar, since the read latency is dominated by Huffman decoding rather than DRAM bandwidth.

Fig. 8(b) presents the corresponding average compression ratio under different configurations. As expected, LZ+Huffman achieves the highest compression ratio at the cost of longer access latency, while LZ-only compression with smaller block sizes results in significantly lower latency but a substantial compression ratio reduction. Fig. 9 further illustrates the average read latency under mixed read/write workloads, with the queue depth set to 8. The results indicate that, as the workload becomes more write intensive, read latency increases. For instance, with LZ-only compression and a 4KB block size, the read latency rises from 397ns to 450ns (DDR5) as the read/write ratio decreases from 100:0 to 50:50. Moreover, when using larger block sizes (2KB and 4KB), the latency difference between DDR4 and DDR5 becomes more evident. The above phenomena are primarily attributed to compression-induced write amplification,

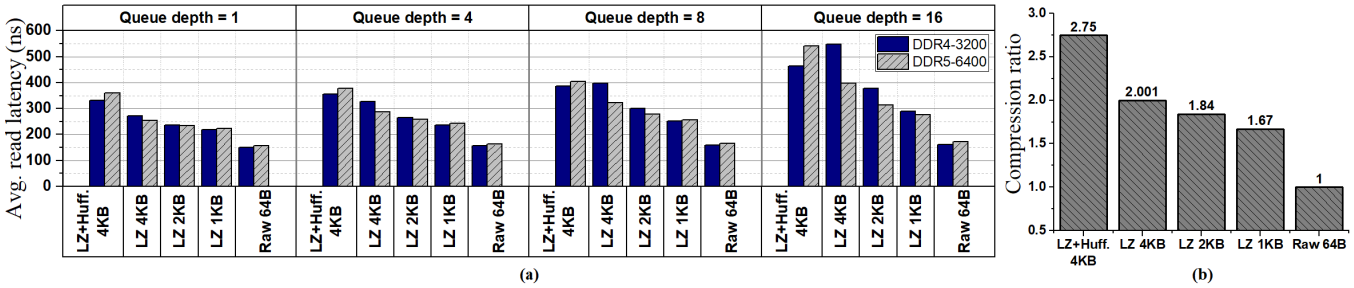

Figure 8: Average random 64-byte read latency and compression ratio under different compression configurations and queue depths.

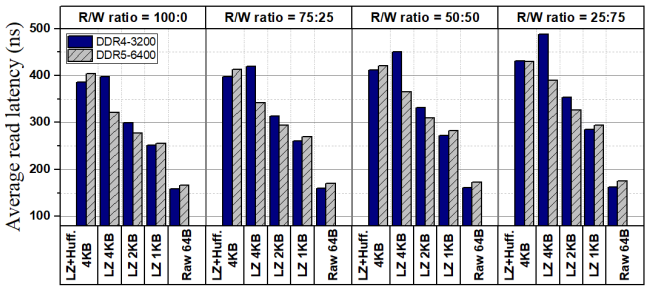

Figure 9: Average random 64-byte read latency in baseline under different read-write ratio mix. The queue depth is 8.

which increases memory bandwidth usage and exacerbates queuing delays.

5.2 Adaptive Compression Granularity

This section presents the results of applying the proposed adaptive compression block chunking technique. As discussed in Section 3.1, this approach aims to optimize the compression ratio vs. speed performance trade-off by selectively reducing the compression block size (i.e., from 4KB to 1KB in this case) only for data whose compressibility is less sensitive to block size reduction. For comparison, we also evaluate a baseline approach where the controller randomly selects data for which the compression block size l_{blk} is reduced from 4KB to 1KB. Fig. 10 and Fig. 11 show the simulation results under different configurations when using LZ-only and LZ+Huffman compression, respectively. The parameter η_{LZ} represents the ratio of 4KB memory pages compressed with l_{blk} = 4KB versus those compressed with l_{blk} = 1KB. The results indicate that the proposed adaptive compression block chunking notably improves the compression ratio vs. speed performance trade-off. By examining sub-figures (a)–(d) and sub-figure (e) in both figures, we observe that, for a given η_{LZ}, the proposed technique achieves a higher compression ratio while reducing read latency compared to the baseline. For example, under a read/write ratio of 100:0 request queue depth of 16, η_{LZ} of 50:50, and DDR5, the proposed design improves the compression ratio from 1.825 to 1.873 while reducing the read latency from 337ns to 318ns.

As shown in Fig. 10 and Fig. 11, when the parameter η_{LZ} increases (i.e., more data are compressed with 4KB blocks), read latency increases due to greater read/write amplification under random 64B accesses. The queuing effect further amplifies this trend, making the read latency advantage of the proposed adaptive compression block chunking more evident at higher request queue depths, as shown in sub-figures (a) and (b) of Fig. 10 and Fig. 11. Additionally, as data accesses include more writes, the read latency benefit of the proposed approach becomes even more significant, as shown in sub-figures (c) and (d) of Fig. 10 and Fig. 11. This is because the proposed technique mitigates write amplification by achieving a better compression ratio, thereby reducing the overall memory traffic burden. Compared to LZ-only compression, the read latency of LZ+Huffman compression is less sensitive to the read/write ratio. This is because Huffman decoding throughput is lower than DDR channel bandwidth, making write amplification effects relatively less impactful in LZ+Huffman compression scenarios.

5.3 Adaptive Entropy Coding Bypass

This section evaluates the effectiveness of adaptive entropy coding bypassing. As discussed in Section 3.1, this approach optimizes the compression ratio vs. speed performance trade-off by selectively enabling entropy coding only for data that benefit the most from complementing LZ search with entropy coding. For comparison, we also evaluate a baseline approach where the controller randomly selects data for which Huffman coding is applied alongside LZ compression.

Fig. 12 and Fig. 13 present the simulated read latency under different configurations, where the throughput of each Huffman decoder is 7.5GB/s and 10GB/s, respectively. The compression block size is fixed at 4KB, and the parameter η_{Huff} represents the ratio of 4KB memory pages compressed with LZ+Huffman versus LZ alone. At a queue depth of 4, as shown in sub-figures (a) and (b) of Fig. 12 and Fig. 13, the read latency advantage of adaptive Huffman coding bypassing is modest. This is because overall read latency is primarily dominated by DRAM internal operations (tRCD and tRP) and the limited throughput of the Huffman decoder. However, as the request queue depth increases to 8, the read latency advantage of adaptive Huffman coding bypassing becomes more significant due to the queuing effect amplifying its impact. Furthermore, in write-heavy workloads (e.g., R/W ratios of 50:50 and 25:75), read latency can even decrease as η_{Huff} increases (i.e., more 4KB memory pages are compressed with LZ+Huffman), particularly in DDR4, as shown in sub-figure (c) of Fig.12 and Fig.13. This effect occurs

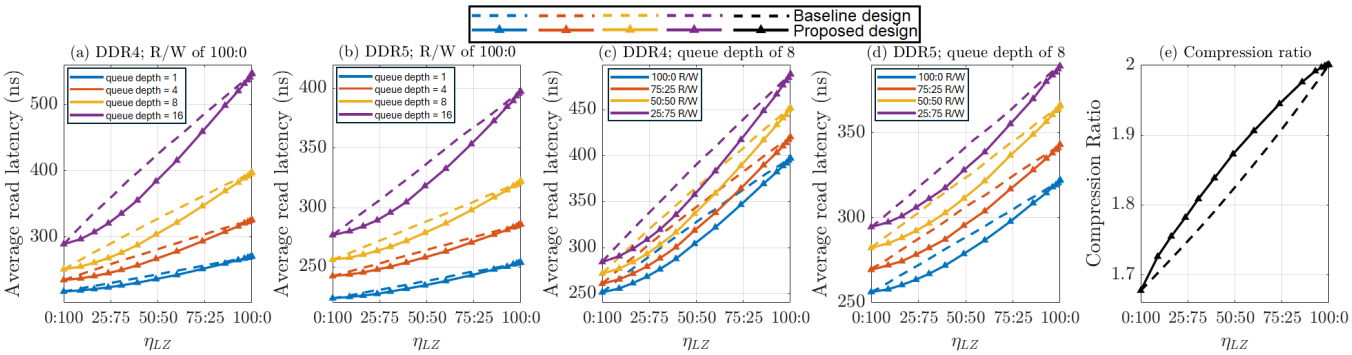

Figure 10: Average read latency and compression ratio under adaptive compression block chunking. η_{LZ} denotes the ratio of 4KB memory pages compressed with l_{blk} =4KB to those with l_{blk} =1KB.

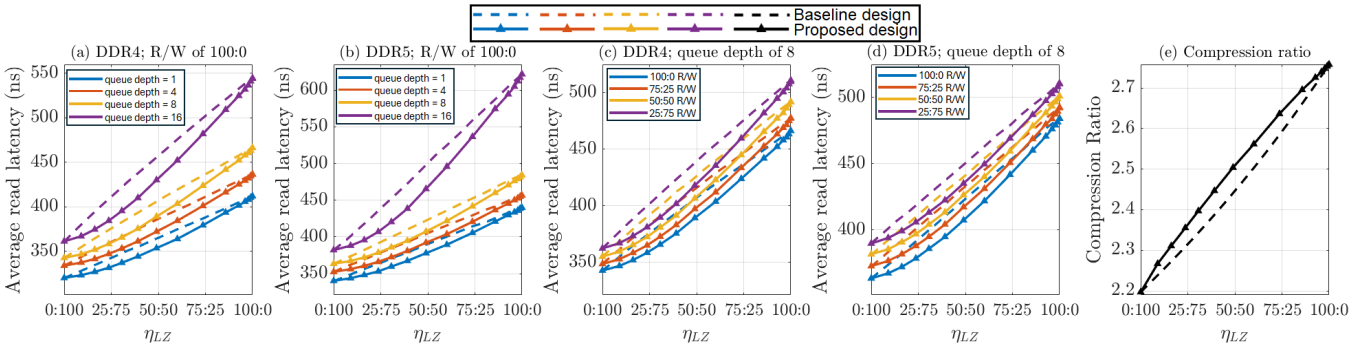

Figure 11: Average read latency and compression ratio under adaptive compression block chunking, with Huffman encoding enabled; Huffman decoder bandwidth is 10GB/s. η_{LZ} denotes the ratio of 4KB memory pages compressed with l_{blk} =4KB to those with l_{blk} =1KB.

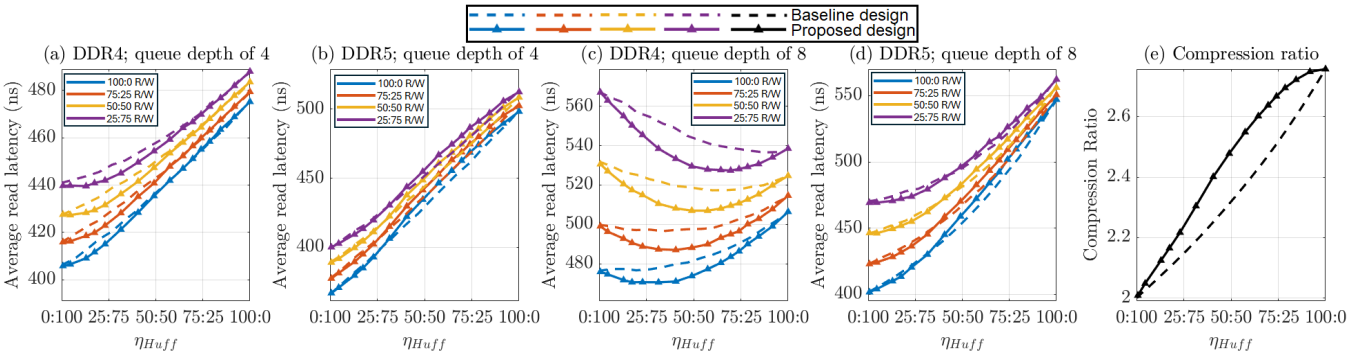

Figure 12: Average read latency and compression ratio under adaptive entropy coding bypass. Huffman decoder bandwidth is 7.5GB/s. η_{LZ} denotes the ratio of 4KB memory pages compressed with LZ+Huffman to those with LZ alone.

because the higher compression ratio enabled by Huffman coding improves DDR bandwidth utilization, which can offset the impact of the Huffman decoder's lower throughput within a certain R/W ratio range. This trend becomes even more pronounced when the Huffman decoder throughput increases, as shown in Fig. 13(c) and (d), where the throughput of the Huffman decoder is 10GB/s.

The sub-figure (e) in both figures demonstrates that adaptive entropy coding bypassing significantly enhances the compression ratio compared to the baseline. This improvement, combined with the results from Fig. 12 and Fig. 13, highlights the effectiveness of the proposed technique in optimizing the trade-off between compression ratio and speed performance. For example, for the 10GB/s Huffman decoder and DDR4, under a request queue depth

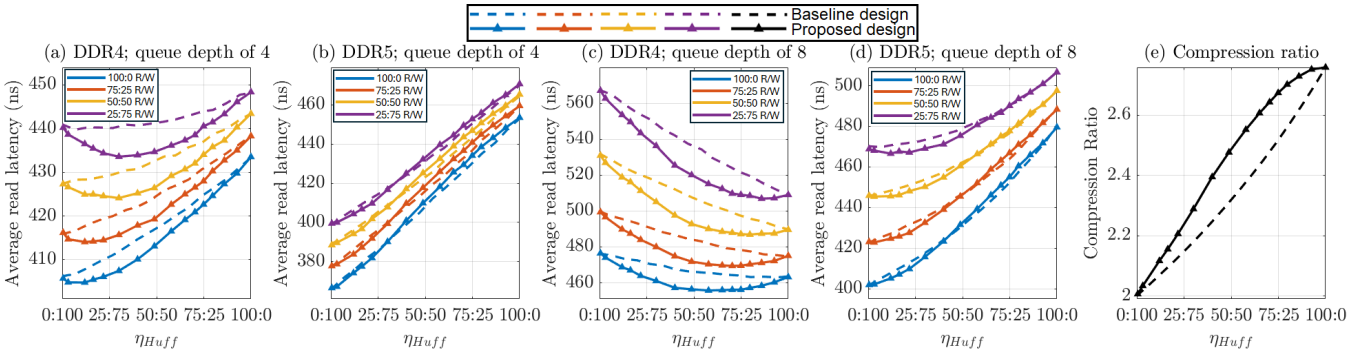

Figure 13: Average read latency and compression ratio under adaptive entropy coding bypass. Huffman decoder bandwidth is 10GB/s. η_{LZ} denotes the ratio of 4KB memory pages compressed with LZ+Huffman to those with LZ alone.

of 8, with η_{Huff} set to 50:50 and an R/W ratio of 50:50, adaptive entropy coding bypassing increases the compression ratio from 2.32 to 2.47 while simultaneously reducing the read latency from 507ns to 492ns compared to the baseline scenario. These results indicate that selectively enabling entropy coding based on data characteristics provides a more efficient balance between compression efficiency and access latency, ultimately enhancing memory bandwidth utilization and reducing queuing delays. Moreover, as workload intensity and queuing effects increase, the benefits of adaptive entropy coding bypassing become more pronounced. The approach not only improves read performance but also minimizes unnecessary processing overhead associated with entropy coding, making it particularly advantageous in high-throughput, mixed read/write workloads.

5.4 Data and ECC Disaggregation

We further evaluate the proposed data-ECC disaggregation approach. As discussed in Section 3.3, the raw DRAM bit error rate is extremely low, resulting in a very low probability (e.g., well below 0.01%) of requiring ECC redundancy for error correction. However, in the rare instances where DRAM errors occur, the read latency will increase due to the additional time required to fetch ECC redundancy. To assess this impact, we examine read latency under both error-free and error-occurring conditions. Additionally, recall that n_c represents the number of CRCs embedded within a compressed block. A higher n_c can reduce the read latency overhead in the event of DRAM bit errors but comes at the cost of increased CRC-induced memory capacity overhead. To explore this trade-off, we evaluate scenarios with $n_c = 1$ and $n_c = 2$.

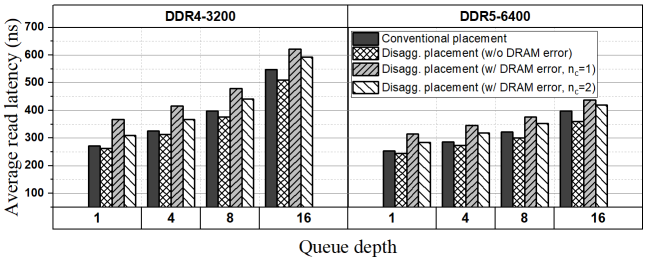

Figure 14: Latency Impact of dis-aggregating data and ECC placement in DIMMs for (a) DDR 4 and (b) DDR 5.

Fig. 14 presents the read latency across four different scenarios under varying request queue depths. For this evaluation, we use LZ-only compression with a 4KB compression block size. As expected, compared to conventional data-ECC placement, the proposed data-ECC disaggregation reduces read latency in the absence of DRAM errors, with its latency advantage further amplified by queuing effects. For example, in DDR5 DRAM, data-ECC disaggregation reduces read latency by 3.2% at a queue depth of 1 and by 9.5% at a queue depth of 16. In the rare event of a DRAM error, the controller detects CRC failures and fetches ECC redundancy from DRAM to perform ECC decoding, leading to an increase in read latency, as shown in Fig. 14. For DDR4, The additional read latency is approximately 104 ns for $n_c = 1$ and 55ns for $n_c = 2$. Under DDR4 DRAM with a queue depth of 8, this corresponds to a read latency overhead of 27% for $n_c = 1$ and 17% for $n_c = 2$.

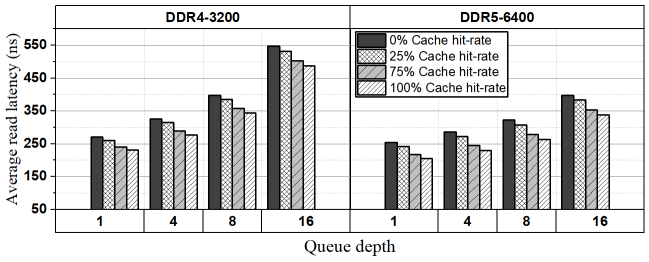

Figure 15: Read latency under different mapping table cache hit rates.

Although the overhead from conditional ECC redundancy fetches is non-negligible, data-ECC disaggregation remains beneficial in most cases due to its ability to reduce latency in error-free conditions, which occur the vast majority of the time. Moreover, increasing n_c provides a trade-off between minimizing error-handling overhead and memory capacity usage, allowing system designers to fine-tune configurations based on workload requirements.

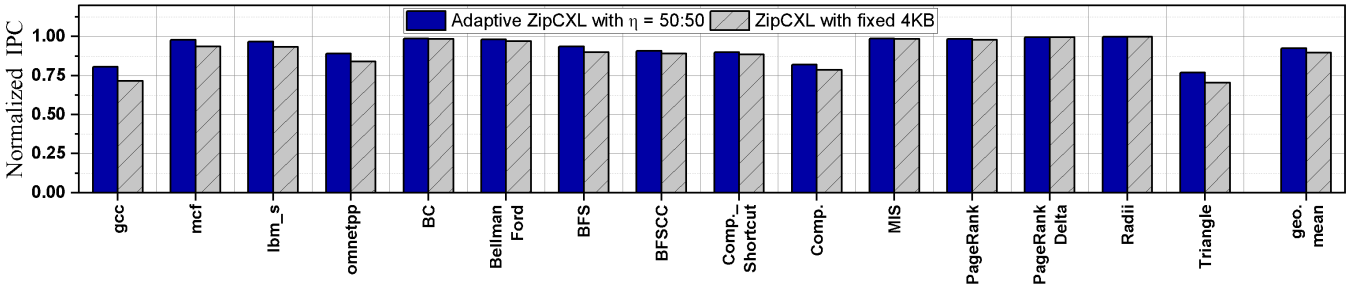

Figure 16: Normalized IPC of ZipCXL under SPEC and Ligra benchmarks.

5.5 Effect of Mapping Table Cache

All the previous simulations were conducted under fully random 64B accesses, rendering the on-chip mapping table cache ineffective. To further investigate its impact, we evaluated scenarios with notable temporal locality in 4KB page accesses, leading to higher mapping table cache hit rates. Fig. 15 presents the read latency across different request queue depths while varying the mapping table cache hit rate. For this evaluation, we use LZ-only compression with a 4KB compression block size. The results indicate that for a fixed cache hit rate, the magnitude of read latency reduction remains consistent across different queue depths. Compared to the zero cache hit rate scenario, mapping table caching significantly reduces read latency. In DDR4, the read latency reduction is approximately 12ns, 37ns, and 50ns for cache hit rates of 25%, 75%, and 100%, respectively. These findings highlight the non-negligible role of mapping table caching in mitigating compression-induced metadata access overhead. By reducing frequent lookups in DRAM-resident mapping tables, caching enhances overall system efficiency, particularly in workloads with strong temporal locality.

5.6 System-level Evaluation

In order to evaluate the performance impact of our proposed techniques operating within a full system, we integrate our ZipCXL simulator with Champsim [13] and measure instructions-per-cycle (IPC) for Ligra [34] and the SPEC CPU 2006 [6] benchmark. As a baseline, we simulate a 12-core CPU system with 8-channel DDR4 and no CXL based memory expansion. For ZipCXL, we set the parameter η_{LZ} to 50:50 so that half of memory pages are compressed with l_{blk} =4KB, and configure roughly 10% of DRAM access hit CXL memory. To highlight the impact of the adaptive compression granularity technique, we compare it with a non-adaptive CXL-based compressed memory configuration in which all pages are compressed with a fixed l_{blk} =4KB granularity.

Fig. 16 shows the IPC results normalized against the baseline. The results indicate a modest to negligible IPC degradation across all benchmarks, with the adaptive ZipCXL solution exhibiting a lower relative IPC degradation. The IPC reduction is primarily due to the increased latency associated with accessing compressed pages on the ZipCXL device. This degradation is most pronounced in memory-intensive benchmarks, which typically have lower raw IPC values; for example, the SPEC CPU 2006 gcc benchmark and the Ligra Triangle benchmark experience normalized IPC degradations of 11% and 9.5%, respectively, when using fixed 4KB LZ compression compared to the adaptive ZipCXL configuration. Overall, the geometric mean of the normalized IPC across a diverse set of workloads demonstrates that incorporating the adaptive ZipCXL system results in a 7.5% decrease in normalized IPC relative to the reference configuration. In contrast, using a fixed 4KB LZ compression scheme yields an additional 3.2% degradation in normalized IPC. It is important to note that higher performance for a given workload can be attained by appropriately tuning the η parameters. Thus, the adaptive ZipCXL approach provides a dynamically tunable trade-off between performance and storage efficiency, enabling compressed main memory expansion with improved adaptability.

5.7 Fused Dual-Stream LZ Compressor

Beyond functional simulation, we evaluated our fused dual-stream LZ compressor (presented in Section 3.1) through RTL-level implementation to assess its hardware efficiency in terms of area and power consumption. For comparison, we implemented both a straightforward design—which utilizes separate 1KB and 4KB LZ compressors—and our fused dual-stream LZ compressor, as illustrated in Fig. 4(a) and (b). Both designs were synthesized using the Cadence Synopsys Design Compiler [36] and the open-source 45nm FreePDK [4]. We extracted area and power consumption estimates to quantify the resource savings enabled by our optimized resource-sharing architecture. The synthesis and power estimation results reveal that, compared to the straightforward implementation, the fused dual-stream design achieves a silicon area reduction of 22% and a power consumption reduction of 36%. These improvements stem from hardware reuse across different compression granularities, eliminating the need for duplicated memory and logic resources. The results show the effectiveness of our architectural optimizations in enabling a more compact and energy-efficient hardware implementation.

6 Related Work

Main Memory Compression: Main memory compression has been extensively studied as a means to increase effective capacity or reduce bandwidth usage, often at the cost of access latency. Many prior works have focused on cacheline granularity compression to minimize latency. Techniques like FPC (Frequent Pattern Compression) [2] and BDI (Base-Delta-Immediate) [29] achieve low-latency compression by exploiting redundancy in small data patterns. These methods have been widely adopted in conjunction with other techniques to improve compression efficiency while preserving performance.

More advanced methods like BPC (Bit-Plane Compression) [17] and zero-aware compression [10] offer better compression ratios for specific data types, but often incur higher decompression latency or require additional architectural modifications, such as extra address translation layers. Similarly, solutions like LCP (Linearly Compressed Pages)[28] and MemZip [33] focus on reducing post-compression address translation overhead or using compression for error correction and energy efficiency rather than capacity gains. Other works, like Transparent Dual Compression [18] and IBM MXT [38], explore larger compression granularities (e.g., 1KB) to balance compression ratio and access latency, particularly in tiered memory systems.

Hardware (De)compressors: Hardware accelerators for lossless compression, such as X-MatchPRO [25] and ALDC [7], have demonstrated the potential for high-speed compression. These accelerators often rely on dictionary-based methods, such as LZ algorithms, combined with techniques like Huffman encoding for efficiency. IBM's NXU compression accelerator [1] further refines LZ compression by using split search windows to balance resource utilization and search quality, achieving compression ratios comparable to GZIP.

7 Conclusion

This paper presents three techniques to enhance compression-capable CXL memory controllers. The first two, adaptive compression block chunking and adaptive entropy coding bypassing, optimize the compression ratio vs. speed performance by dynamically adjusting compression configurations. The third, data-ECC disaggregation, improves effective DRAM bandwidth utilization, mitigating compression-induced read/write amplification. To evaluate these techniques, we developed a simulation platform to analyze compression-performance trade-offs and conducted RTL-level design and synthesis to assess silicon cost overhead. The results demonstrate that the proposed techniques significantly improve the trade-off between compression ratio and speed performance with minimal hardware overhead, making them well-suited for scalable, low-latency CXL memory expansion.

References

[1] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J Starke, Haren Myneni, and Charlie Wang. 2020. Data compression accelerator on IBM POWER9 and z15 processors: Industrial product. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 1–14.

[2] Alaa R Alameldeen and David A. Wood. 2004. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. *Technical Report, Computer Sciences Dept., UW-Madison* (2004).

[3] Amazon. 2025. *Opensource datasets from Amazon AWS.* Retrieved February 15, 2025 from https://registry.opendata.aws/

[4] Electronic Design Automation at North Carolina State University. 2025. *45nm free PDK.* Retrieved February 15, 2025 from https://eda.ncsu.edu/freepdk/freepdk45/

[5] Yann Collet. 2021. RFC 8878: Zstandard Compression and the 'application/zstd' Media Type.

[6] Standard Performance Evaluation Corporation. 2006. *SPEC CPU® benchmark.* https://www.spec.org/cpu2006/

[7] David J Craft. 1998. A fast hardware data compression algorithm and some algorithmic extensions. *IBM Journal of Research and Development* 42, 6 (1998), 733–746.

[8] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *Comput. Surveys* 56, 11 (2024), 1–37.

[9] Sebastian Deorowicz. 2014. Silesia compression corpus. *Silesia University* (2014). https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia

[10] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *International Symposium on Computer Architecture (ISCA)*. 74–85.

[11] Michael J. Freedman. 2000. The Compression Cache: Virtual Memory Compression for Handheld Computers. https://api.semanticscholar.org/CorpusID:59821443

[12] Jean-Loup Gailly and Mark Adler. 2004. Zlib compression library. *Apollo - University of Cambridge Repository* (2004).

[13] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jiménez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. *The Championship Simulator: Architectural Simulation for Education and Competition.* https://doi.org/10.48550/arXiv.2210.14324

[14] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[15] Marvell Technology Inc. 2025. *CXL Near-Memory Compute and Expansion.* Retrieved February 15, 2025 from https://www.marvell.com/products/cxl.html

[16] Kaggle. 2025. *Open Datasets from Kaggle.* Retrieved February 15, 2025 from https://www.kaggle.com/datasets

[17] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 329–340.

[18] Seikwon Kim, Seonyoung Lee, Taehoon Kim, and Jaehyuk Huh. 2017. Transparent dual memory compression architecture. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 206–218.

[19] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 574–587.

[20] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[21] Yiwei Li and Mingyu Gao. 2023. Baryon: Efficient hybrid memory management with compression and sub-blocking. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 137–151.

[22] Tobias Mann. 2022. *Just How Bad Is CXL Memory Latency?* Retrieved February 15, 2025 from https://www.nextplatform.com/2022/12/05/just-how-bad-is-cxl-memory-latency/

[23] Micron Technology, Inc. 2023. *Micron memory expansion module using CXL.* Retrieved February 15, 2025 from https://www.micron.com/products/memory/cxl-memory

[24] Brian Morris and Prakash Chauhan. 2024. *Making Memories at HyperScale with CXL.* Retrieved February 15, 2025 from https://computeexpresslink.org/wp-content/uploads/2024/10/CXL_Q3-Webinar_Making-Memories-at-HyperScale-with-CXL_FINAL.pdf

[25] Jose Luis Nunez, Simon Jones, and Stephen Bateman. 2001. X-MatchPRO: a high performance full-duplex lossless data compressor on a ProASIC FPGA. In *Proceedings of the International Workshop on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*. 56–60.

[26] Keshab K Parhi. 1992. High-speed VLSI architectures for Huffman and Viterbi decoders. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39, 6 (1992), 385–391.

[27] Keshab K Parhi. 2007. *VLSI digital signal processing systems: design and implementation.* John Wiley & Sons.

[28] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*. 172–184.

[29] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the international conference on Parallel architectures and compilation techniques*. 377–388.

[30] Quandl. 2025. *Quantative dataset source from Quandl.* Retrieved February 15, 2025 from https://data.nasdaq.com/publishers/QDL

[31] Samsung Electronics Co., Ltd. 2024. *Samsung CXL memory solutions.* Retrieved February 15, 2025 from https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/

[32] Satyabrata Sarangi. 2022. *Hardware Architectures for Lossless Compression.* Ph. D. Dissertation. University of California, Davis.

[33] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 638–649.

[34] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[35] SK Hynix Inc. 2024. *SK Hynix Presents CXL Memory Solutions Set to Power the AI Era at CXL DevCon 2024.* Retrieved February 15, 2025 from https://news.skhynix.com/sk-hynix-presents-ai-memory-solutions-at-cxl-devcon-2024/

[36] Synopsys, Inc. 2025. *Synopsys Design Compiler Webpage.* Retrieved February 15, 2025 from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[37] Transaction Processing Performance Council (TPC). 2025. *TPC-H benchmark.* Retrieved February 15, 2025 from https://www.tpc.org/tpch/

[38] R Brett Tremaine, Peter A Franaszek, John T Robinson, Charles O Schulz, T Basil Smith, Michael E Wazlowski, and P Maurice Bland. 2001. IBM memory expansion technology (MXT). *IBM Journal of Research and Development* 45, 2 (2001), 271–285.

[39] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

[40] Darko Zivanovic, Pouya Esmaili Dokht, Sergi Moré, Javier Bartolome, Paul M Carpenter, Petar Radojković, and Eduard Ayguadé. 2019. DRAM errors in the field: A statistical approach. In *Proceedings of the International Symposium on Memory Systems*. 69–84.