CXL Main Memory Compression aware Tiered Cuckoo Hash Map

Benjamin Herman
Troy, New York
benherman345@gmail.com

***Abstract*—Current cuckoo hash tables access sparse and dense regions uniformly during insertion and eviction. Under CXL hardware-compressed memory, each access fetches an entire 1-4 KB compression block regardless of occupancy. This paper proposes a Tiered Cuckoo Hash Map that partitions the table into multiple tables of exponentially increasing size. Insertions begin in the smallest table and propagate to larger tables only through evictions, concentrating frequently accessed keys in small, cache-resident tables while colder entries spill into larger, compressed tables. This structure reduces compressed memory accesses during lookups and avoids full-table rehashing by adding new tables as the structure grows.**

I. Introduction

Compute Express Link (CXL) enables large, hardware-compressed main memory by moving compression into the memory device and exposing it transparently to software. Under current CXL specifications, memory is compressed and accessed in coarse-grained blocks (1–4 KB), with blocks remaining compressed across accesses. While compression increases effective capacity, it also changes the cost model of memory access: individual accesses may trigger large block transfers and incur higher latency. As a result, data structures designed for uniform, uncompressed DRAM no longer behave optimally under CXL-style memory.

Zhang and coauthors [1] study this shift for hash tables by reanalyzing chained and Cuckoo hashing under hardware-compressed memory. Their work shows that block-oriented hash tables can leave logical space unused while relying on compression to reclaim physical capacity. In blocked Cuckoo hashing, buckets are aligned to compression blocks, improving compressibility and reducing metadata overhead. However, this approach treats the hash table as a single flat eviction domain. As the table grows, sparsely populated regions are accessed as frequently as dense ones during insertions and evictions, amplifying compressed-block reads and writes. Compression improves capacity, but does not reduce the scope of eviction-induced memory traffic.

This paper addresses that limitation by restructuring the Cuckoo hash table itself. We propose a Tiered Cuckoo Hash Map composed of multiple tables of increasing size. Insertions begin in the smallest table and propagate to larger tables only through cuckoo-style evictions. Recently inserted or frequently accessed keys are more likely to reside in smaller, denser tables that fit in fast, uncompressed memory, while colder entries spill into larger tables benefiting from hardware compression. Growth is handled by adding new tables rather than resizing existing ones, avoiding table-wide scans of compressed memory.

We describe the design of the Tiered Cuckoo Hash Map and experimentally evaluate its lookup throughput, insertion throughput, load factor, and compression behavior relative to prior compression-aware Cuckoo hashing designs.

II. Design

The Tiered Cuckoo Hash Map organizes data across a sequence of tables $T_1, ..., T_n$, where each table T_i has size $t_s \cdot t_m^{i-1}$ (refer to TABLE I for symbols). Insertions begin at T_1 and follow standard cuckoo hashing within that table. When all candidate buckets in T_1 are full, an existing entry is evicted and reinserted into T_2. This eviction process continues through the tiers, with displaced entries moving from T_i to T_{i+1} until a placement succeeds. When an entry is displaced from T_n, it cycles back to T_1, maintaining a continuous eviction chain across all tables.

All tables share the same hash function. The redistribution of keys across tiers occurs naturally because each table has a different size: hashing a key produces the same value, but taking that value modulo a larger table size yields a different bucket index. This eliminates the need to compute multiple hash functions or maintain separate hash state for each tier. When the structure approaches its load threshold, it scales by adding a new table T_{n+1} of size $t_s \cdot t_m^n$ rather than rehashing existing entries. The new table is initially empty and sparse, making it highly compressible in hardware-compressed memory.

The hierarchy exploits the access patterns typical of cuckoo hashing. Recently inserted keys remain in T_1 unless evicted, and frequently accessed keys tend to be found in earlier lookups before searching deeper tables. Since T_1 is small—only t_s buckets—it fits entirely in cache and avoids main memory access latency. Subsequent tables grow exponentially: T_2 is t_m times larger, T_3 is t_m^2 times larger, and so on. This exponential growth means that most keys reside in the larger tables which exceed cache capacity and reside in main memory. However, because these tables are sparsely populated during normal operation, they compress efficiently when stored in CXL memory. Each bucket occupies multiple cache lines, so even a single lookup to a large table incurs the cost of fetching a full compression block. By concentrating lookups in the smaller tables, the structure reduces the frequency of these expensive compressed memory accesses.

III. Methods

To test the effectiveness of this hash map, I implemented it in C++ [2]. This object was created with predefined parameters, and I used data that could be read. To aid evaluation I benchmarked my results against those of Zhang, adopting their parameters. The complete list of parameters is:

TABLE I

Symbol	Definition	Value
t_s	Number of buckets in the table T_1	1024
t_m	Table scale amount per i increment	4x
S_{kv}	Size of a single KV pair (8B key)	64B
n_b	Number of KV pairs per bucket	5 KV
S_b	Total size of a bucket	264B
n_{kv}	Total number of KV pairs	
m_{bck}	Number of buckets	
α	Load factor	$\frac{n_{kv} \cdot S_{kv}}{m_{bck} \cdot S_b}$

The implementation uses a fixed initial table size of 1024 buckets for T_1, with each subsequent table scaled by a factor of 4. This scaling factor determines how quickly the hierarchy expands: T_2 contains 4096 buckets, T_3 contains 16384 buckets, and T_4 contains 65536 buckets. Each bucket stores exactly 5 key-value pairs, where keys are 8 bytes and each complete KV pair occupies 64 bytes, resulting in a 264-byte bucket size including metadata. The hash function used is C++'s std::hash applied uniformly across all tables, with the bucket index determined by taking the hash value modulo the current table size.

For initialization, I inserted a minimum of 120000 uniformly random 8-byte integer keys to ensure the structure contains at least four tables (T_1 through T_4). Additional elements were inserted beyond this threshold to test the structure at various load factors. Statistics were collected only after the fourth table was created to eliminate noise from the initial growth phase. Each insertion attempt follows the standard cuckoo eviction process: if all candidate buckets in T_1 are full, an existing entry is evicted and reinserted into T_2, continuing through the tiers until a placement succeeds or the eviction cycles back to T_1.

Lookup throughput was measured by performing 10,000 random queries against the populated hash map and dividing the total number of lookups by the elapsed time in seconds. Insert throughput was measured similarly by timing 10,000 insertions of new random keys and calculating operations per second. Both measurements were repeated across different load factors, ranging from 0.1 to 0.6, to observe how performance degrades as the structure fills.

To evaluate compression behavior, each table was divided into fixed-size memory blocks matching typical CXL compression granularities of 1024 bytes and 4096 bytes [3]. Each block was compressed independently using the LZAV algorithm [4], which provides fast compression suitable for hardware implementation. The compression ratio for each table was calculated by summing the sizes of all compressed blocks and dividing by the original uncompressed table size. This block-level compression mimics how CXL memory controllers operate, compressing and decompressing data in fixed-size chunks rather than treating the entire table as a single compressible unit.

To simulate the latency penalty of accessing compressed memory, I added 200 nanoseconds to the measured time whenever the benchmark accessed a memory block expected to be stored in compressed form. This penalty reflects the additional time required to decompress a block on a CXL memory access compared to reading from standard DRAM. Blocks were considered compressed if they resided in tables large enough to exceed typical cache sizes, specifically T_4. Smaller tables T_1, T_2, and T_3 were assumed to remain cache-resident and uncompressed during normal operation, incurring no additional latency penalty.

IV. Results

The Tiered Cuckoo Hash Map achieves insert throughput between 2.231 and 1.586 M ops/s across load factors from 0.11 to 0.61, with lookup throughput ranging from 30.670 to 5.369 M ops/s over the same interval. Peak performance occurs at low load factors and performance degrades as load factor increases. Compression block size has minimal impact on throughput.

The compression ratio, shown in Figure 1, increases linearly with load factor, rising from approximately 0.25 at a load factor of 0.11 to 0.65 at a load factor of 0.61. This trend indicates that as the hash map fills, the compressed size grows proportionally to the amount of data stored, approaching the size of the uncompressed structure. The compression ratio measures the size of the compressed tables divided by their uncompressed size: a ratio of 0.25 means the compressed data occupies 25% of the original space, while a ratio of 0.65 means it occupies 65%. Both 1 KB and 4 KB compression blocks follow nearly identical trajectories, with 4 KB blocks compressing marginally better throughout the load factor range.

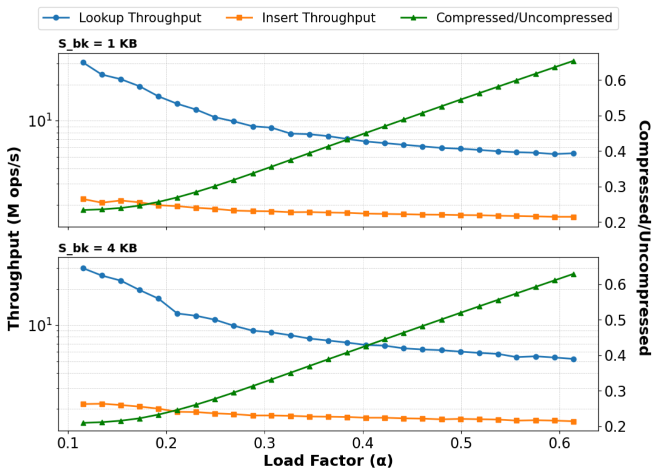

Fig. 1. Tiered Cuckoo load factor vs compression ratio

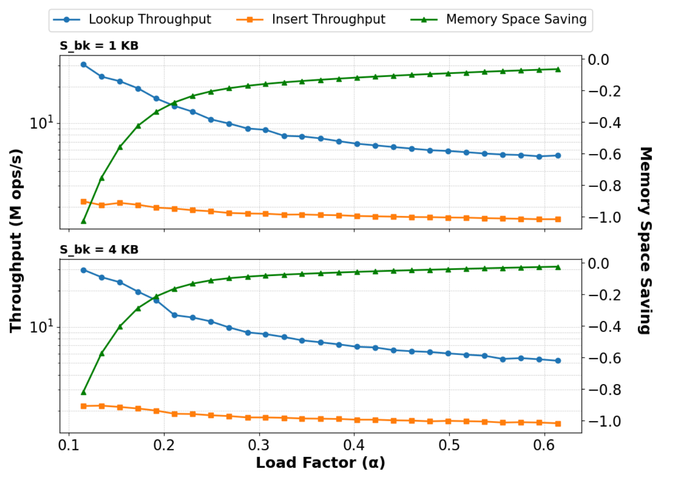

Fig. 2. Tiered Cuckoo load factor vs memory savings

Zhang's performance for a regular cuckoo map.

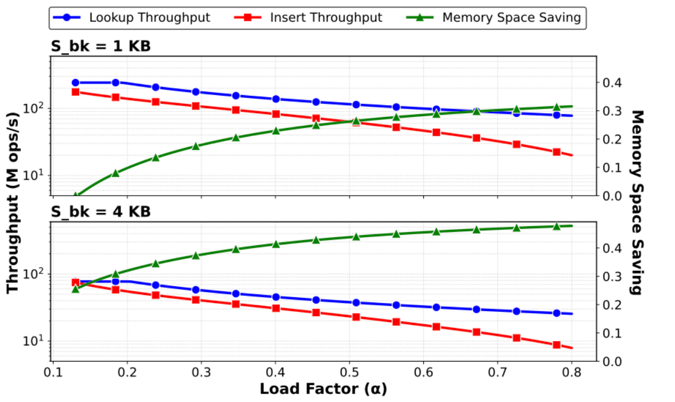

Fig. 3. Zhang's Cuckoo load factor vs memory savings [1]

Figure 2 presents memory space savings, described in Zhang's paper it is calculated as $1 - \left(\frac{\text{compressed size}}{\text{KV data size}}\right)$, where KV data size refers only to the 64 bytes per key-value pair without any metadata or bucket overhead. Negative values indicate that the compressed hash map structure exceeds the size of the raw key-value data alone. At low load factors, memory space savings are deeply negative, reaching approximately −0.9, meaning the compressed structure is 1.9 times larger than the raw data. As load factor increases to 0.61, savings improve to approximately −0.05, indicating the compressed structure is only 5% larger than the raw data size. This improvement occurs because bucket metadata and padding constitute a smaller fraction of total memory as more entries fill the available space.

The negative memory savings do not indicate compression failure. Instead, they reflect the overhead inherent in maintaining bucket structures with metadata and alignment padding. Each bucket stores 5 key-value pairs plus metadata, and buckets must align to compression block boundaries. At low load factors, most buckets are partially empty, so metadata overhead dominates. The 56 bytes of value data in each key-value pair were filled with uniformly random bytes during testing, which are effectively incompressible. Therefore, the compression algorithm can only reduce the size of empty bucket slots and metadata, not the actual key-value data itself. As the hash map fills and fewer empty slots remain, the ratio of compressed structure size to raw data size approaches 1.0, causing memory space savings to approach zero.

Comparing these results to Zhang's blocked cuckoo hashing shown in Figure 3, the memory behavior differs substantially. Zhang's approach achieves positive memory space savings, ranging from near 0% at low load factors to approximately 25-40% at higher load factors. This indicates Zhang's compressed structure is smaller than the raw key-value data size, suggesting their implementation either compresses the data itself. The Tiered Cuckoo Hash Map prioritizes a different design goal: concentrating recently accessed entries in small, cache-resident tables while allowing larger, sparser tables to benefit from hardware compression. This structure reduces the number of compressed memory block fetches during typical lookup operations, trading higher metadata overhead for fewer expensive decompression events.

V. Conclusion

The experimental evaluation demonstrates that the Tiered Cuckoo Hash Map maintains stable insert throughput across a wide range of load factors while lookup throughput degrades predictably as the structure fills. The compression ratios achieved range from 0.25 to 0.65, indicating the structure compresses to 25-65% of its uncompressed size by exploiting sparsity in the larger tables. This compression behavior is effective given that the 56 bytes of value data in each entry consist of uniformly random bytes that cannot be compressed at all.

The comparison to Zhang's results reveals a limitation in the evaluation methodology. The test data used incompressible random values while Zhang assumed compressible data. Future work should evaluate using compressible datasets to measure whether the tiered structure provides additional compression benefits beyond metadata reduction.

The primary challenge is lookup latency when keys reside in larger tables. As load factor increases and more keys propagate to the back tables, lookup throughput degrades from because the structure must check each tier sequentially. Individual lookups that miss the front tables incur the full cost of searching through multiple compressed blocks.

References

[1] T. Zhang, " Rethinking In-Memory Hash Table Design for CXL-Based Main Memory Compression ," *IEEE Computer Architecture Letters*, vol. 24, no. 2, pp. 357–360, July 2025, doi: 10.1109/LCA.2025.3628805.

[2] B. Herman, "ACS Final Project." [Online]. Available: https://github.com/ItchyTrack/ACS-final-project

[3] P. Chauhan, C. Petersen, B. Morris, and J. Glisse, "Hyperscale Tiered Memory Expander Specification – for Compute Express Link (CXL)," Technical Specification Version 1.0, Revision 1, Oct. 2023. [Online]. Available: https://www.opencompute.org/documents/hyperscale-tiered-memory-expander-specification-for-compute-express-link-cxl-1-pdf

[4] Avaneev, "lzav." [Online]. Available: https://github.com/avaneev/lzav