CLX Main Memory Compression aware Tiered Cuckoo Hash Table

Benjamin Herman
Troy, New York
benherman345@gmail.com

***Abstract*—Curent Cuckoo Hashs do not take advantage of CLX Main Memory Compression. In this paper, I proprose a restructuring the multiple tables in a Cuckoo Hash to take advantage Main Memory Compression. This Tiered Cuckoo Hash Table keeps data densely packed in smaller tables to allow larger compressed tables to take extra values.**

I. Introduction

A. Paper overview

II. Design

The Tiered Cuckoo Hash Table organizes data across a sequence of tables $T_1, ..., T_n$, where each table T_i has size $t_s \cdot t_m^i$. Insertions begin at T_1; displaced entries move to T_2 and continue through the tiers. When an entry is displaced from T_n, it cycles back to T_1, preserving a continuous cuckoo-style eviction chain across all tables. All tables share the same hash function, and the reshuffling of keys is due to the changing table size. When the structure nears its load threshold, it scales by adding a new table $T_{\{n+1\}}$ rather than rehashing existing entries.

The design of the Tiered Cuckoo Hash Table prioritizes efficient memory access and cache utilization. Smaller tables at the top of the hierarchy are densely populated, increasing the likelihood that lookups will find their target in the first table checked. By keeping these tables compact, reads and writes can be performed quickly, benefiting from fast cache hits and minimal memory overhead. Larger tables deeper in the hierarchy are much sparser and reside in main memory, where compression is applied automatically by the hardware. These tables are accessed less frequently, so their storage format does not significantly impact performance. All tables use the same hash function, avoiding the need to compute a new hash for each tier. The structure scales by adding new tables rather than resizing and rehashing existing ones. Each new tier is initially sparse and can be efficiently stored in hardware-compressed memory. As tables increase in size across the hierarchy, the existing hash function continues to distribute keys appropriately without modification.

III. Methods

To test the effictiveness of this hash map I impelmetned it in C++ [1]. This object was created with parameters and were set before hand and data that I could read. To aid evaluation I benchmarked my results against those of Zhang and I adopt their parameters. [2]. The complete list of parameters are:

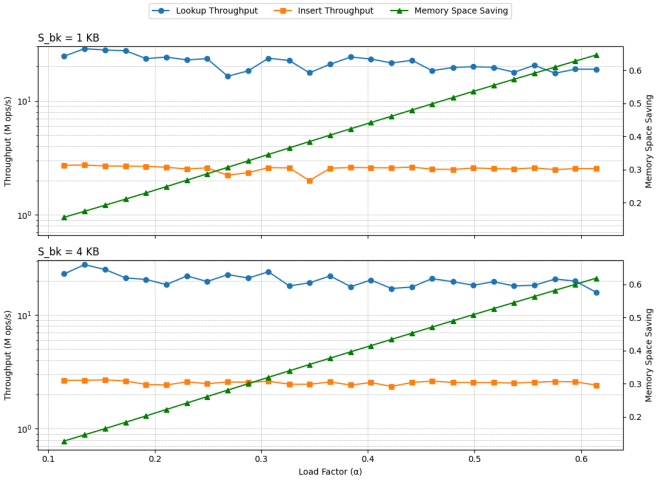
Symbol	Definition	Value
t_s	Number of buckets in the table T_0	1024
t_m	Table scale amount per i increment	5x
S_{kv}	Size of a single KV pair (8B key)	56B
n_b	Number of KV pairs per bucket	5 KV
S_b	Total size of a bucket	328B
n_{kv}	Total number of KV pairs	
m_{bck}	Number of buckets	
α	Load factor	$\frac{n_{kv} \cdot S_{kv}}{m_{bck} \cdot S_b}$

I benchmarked lookup throughput and insert throughput and recored the load factor at this point. I also recored the compression ratio splitting each table into small blocks that each were compressed by LZAV [3]. The sum of the sizes of these compressed blocks divded by the original size of the table gives the compression ratio. The splitting into small blocks is to simulate memory block compression.

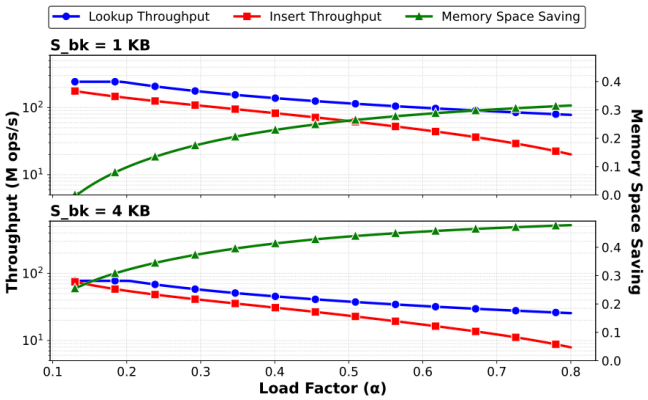
I used the same two memory block sizes as in the other papar [2], 1024B and 4096B

IV. Conclusion

My simulated performance for a Tiered Cuckoo.



Zhang's performance for a regular cuckoo map.



References

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

[2] 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.

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