Tiered Hash Map

December 15, 2025

Joyal Mathew

1. Introduction

Modern in-memory key-value stores (such as Redis) typically assume uniform memory access to DRAM. Now, as SSD performance improves and CXL powered compressed memory becomes viable, this assumption fails. Storing solely in DRAM leaves optimizations on the table as data structures may more efficiently be represented across multiple memory hierarchies. This project will investigate this idea as it applies to hash maps and introduce a new tiered-memory hash map data structure. We will investigate the effect of placement policy on throughput and memory usage.

2. Literature Review

In a *TMO: Transparent Memory Offloading in Datacenters*, Johannes Weiner et al. investigated techniques for transparently offloading memory in heterogeneous hierarchies at a kernel level [1]. They build a system called TMO that can dynamically determine the ideal amount of memory to offload based on workload and hardware characteristics. They also implemented efficient kernel routines to actually offload memory to different hardware devices. Their placement policy used stall pressure to dynamically determine which memory needs to be in RAM. This allows the system to quickly respond to a volatile program. Stall pressure is a good metric when working at the OS level where very few assumptions can be made about memory usage. However, when constructing a tier-aware data structure, still pressure is too slow an indicator. When threads are stalling on memory access, the problem is already there. Using specific information about the data structure being implemented can allow for even more responsive placement policies.

In *BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory*, Joy Arulraj et al. introduce BzTree, a lock-free B-tree implementation for NVM [2]. To achieve this, they rely on persistent multi-word compare-and-swap operations. The idea is to use tracked descriptors when performing atomic NVM operations. This allows for things like setting a dirty bit so that other threads know to flush prior to reading. This concept could certainly be useful in its application to CXL compressed memory which is similar to NVM in being slow.

3. Implementation

3.1. Memory Simulation

Due to the nature of compressed memory, it cannot be directly written to and from. Compression requires a large context, and writing directly into a compressed memory block is a difficult task. As such, compressed memory must be interacted with through explicit acquire and flush operations at the block-level that compress/decompress on the fly.

To simulate this memory hierarchy in user-space with no hardware changes, the three levels of memory are represented differently. RAM is represented with a chunk of memory acquired through a call to `malloc`. CXL compressed memory is represented also with a large chunk of memory but is additionally partitioned into smaller chunks which are individually compressed when writing a block and decompressed when reading a block. The LZ4 compression algorithm is used in this project [3]. SSD is represented with a chunk of memory backed by a file and bypassing the page cache using `msync/mprotect`.

Access to all three types of memory are made symmetric from an API standpoint because memory is only interacted with on a block level. This greatly simplifies implementation of the memory system but complicates the writing of application code. This trade-off, however, is acceptable in the limited scope of this project.

The memory allocator, instead of returning raw pointers, returns handles which define a memory tier and a chunk within that tier. Operations such as acquire, flush, and migrate (as shown in Figure 1) are provided on these handles which allows application code to use memory without needing to know what tier it is in.

```
Ptr ta_migrate(TieredAllocator *ta, Ptr ptr, MemoryTier tier);
void *ta_acquire(TieredAllocator *ta, Ptr ptr);
void ta_flush(TieredAllocator *ta, Ptr ptr);
```

Figure 1: Example memory operation headers

3.2. Tiered Hash Map

Implementing a hash map across multiple memory hierarchies has a fundamental issue due to the nature of the data structure. Hash maps are meant to perform pseudo-random indexing regardless of key similarity. This means that they struggle from having zero spacial locality. Having accessed a particular element in a hash map says nothing about whether neighboring elements will be accessed soon. Despite this, in certain applications, hash maps benefit from the property of temporal locality. Consider an in-memory cache for something like incoming API requests. Having accessed a particular element indicates that that same element may be accessed again in the future because that is the very situation in which a cache useful.

In many caching applications, incoming request frequencies are likely to follow power law distributions [4] or similar light-tailed distribution. This means that there can potentially be performance gained from keeping common items in lower latency memory.

Figure 2 describes the memory layout of the tiered hash map proposed in this project. Like a traditional hash map, we keep a linear array of buckets which can be indexed into with a hash value. Each bucket stores a pointer to the head of a linked list of "chunks". Each chunk is a region of tiered memory that is has a fixed size set by the memory system. Each chunk may hold multiple entries (key-value pairs). Accessing an element requires searching through each chunk of a particular bucket. This can be quite costly if the size of the linked lists is high. However, in that case there is a trade-off with memory usage since each bucket may be stored in any tier of memory.

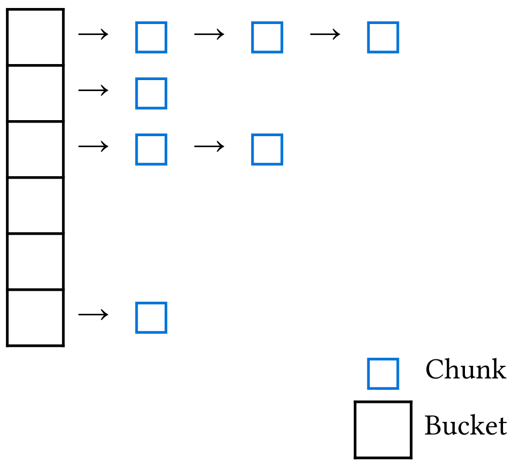

Figure 2: Memory diagram of the tiered hash map

In this project we will implement a hash map that maps string keys to 64 bit integer values. Keys will be hashed using MurmurHash [5]. The buckets will also be pre-allocated and never resized to simplify implementation. However, different load ratios will be investigated.

3.2.1. Placement Policy

The key question for this data structure is that of placement policy. How will the algorithm choose which buckets are stored in which tier? Note that it doesn't necessarily make sense to store multiple chunks within bucket in different memory tiers due to the linear scan requirement. If the last chunk in a list benefits from being stored in RAM but every chunk before it is stored in SSD, there will be little speedup observed due to the requirement of a linear scan through those SSD chunks to get to it.

In this project, our RAM placement policy is as follows: a fixed number of buckets will be designated as RAM-resident. This will be treated like an LRU cache where previously used elements will be kept and unused elements will be evicted to either being CXL-resident or SSD-resident. To approximate, with low overhead, an LRU cache, this implementation will use the SIEVE algorithm [6].

4. Results

4.1. Memory Simulation

The memory tiers were evaluated as implemented to analyze latency and found to approximate the memory hierarchy of RAM ⟶ CXL ⟶ SSD.

	Read Latency (ns ± 95% CI)	**Write Latency (ns ± 95% CI)**
RAM	28.74 ± 1.35	27.46 ± 0.17
CXL	89.69 ± 1.47	199.75 ± 1.10
SSD	168.33 ± 2.25	2151.64 ± 895.65

Figure 3: Memory system performance

4.2. Input Data

To simulate a distribution similar to requests coming into an in-memory cache, we will task the hash map with the simple task of counting work frequencies in English text. This should also have a light-tailed distribution.

4.3. Ram Capacity

We shall first investigate the effect of RAM capacity on throughput and memory usage. The tiered hash map pre-allocates buckets and has a pre-determined number of buckets to be used as RAM (RAM capacity). Allocating more buckets to RAM should results in higher throughput but also higher memory usage.

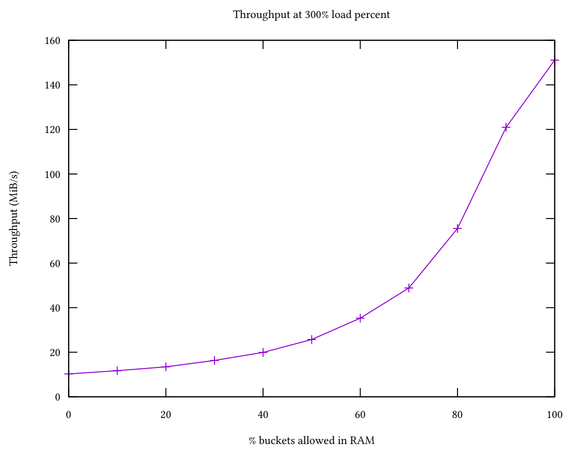

Figure 4: Throughput vs % RAM @ 300% load

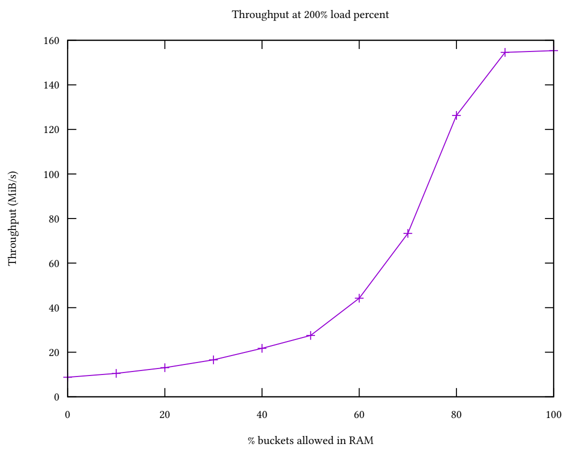

Figure 5: Throughput vs % RAM @ 200% load

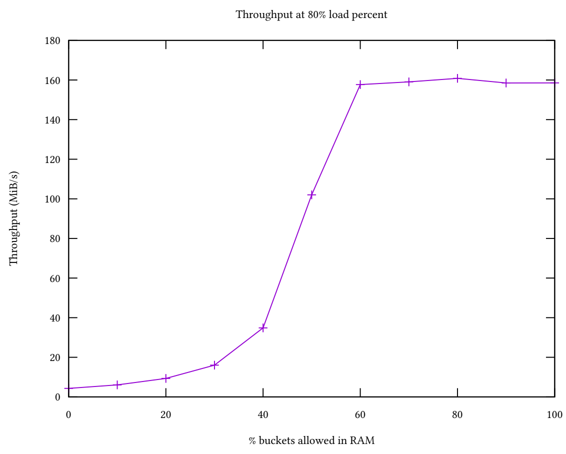

Figure 6: Throughput vs % RAM @ 80% load

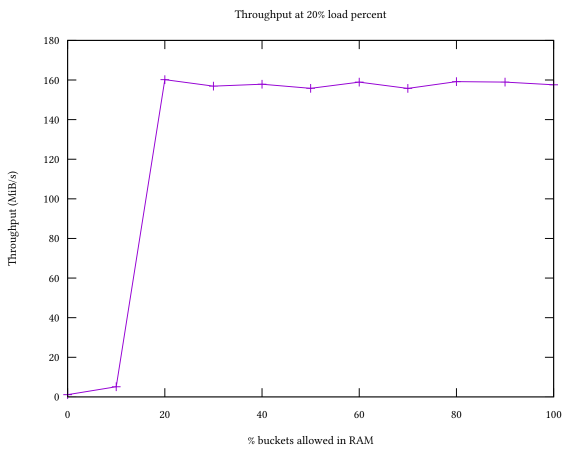

Figure 7: Throughput vs % RAM @ 20% load

As expected, allocating more buckets to RAM results in higher throughput. In fact, the throughput seemingly increases exponentially and hits a cap when RAM can store every element. Now we can look at memory usage.

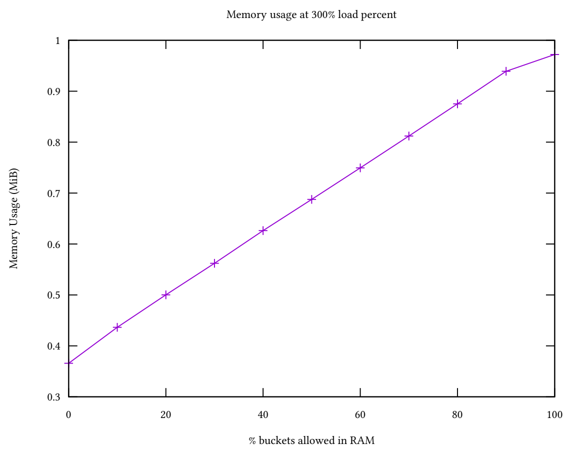

Figure 8: Throughput vs % RAM @ 300% load

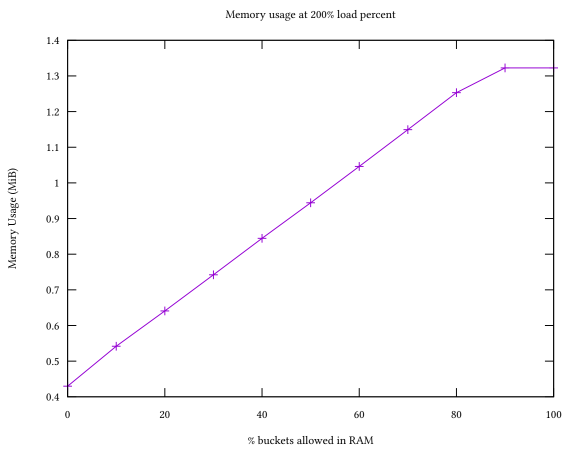

Figure 9: Throughput vs % RAM @ 200% load

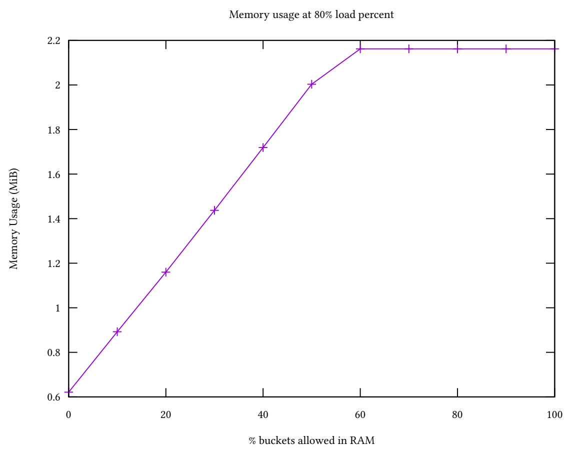

Figure 10: Throughput vs % RAM @ 80% load

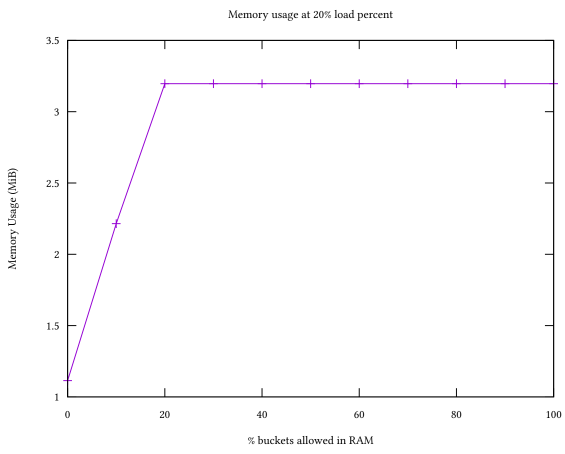

Figure 11: Throughput vs % RAM @ 20% load

Again, as expected, memory usage increases as more buckets are stored in RAM. For memory, the increase is linear, again hitting a ceiling when every element is stored in RAM.

5. Conclusion

In this project, we successfully implemented a framework for user-space simulation of tiered memory and used it to implement and benchmark the tiered hash map data structure. Placement policy was investigated and it was found that choosing the number of buckets to allocate for RAM is a trade-off between throughput and memory usage. This parameter should be tuned based on the application.

Bibliography

[1] J. Weiner *et al.*, “TMO: Transparent memory offloading in datacenters,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 609–621.

[2] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, “Bztree: A high-performance latch-free range index for non-volatile memory,” *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[3] [Online]. Available: https://github.com/lz4/lz4

[4] A. Arampatzis and J. Kamps, “A study of query length,” in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, 2008, pp. 811–812.

[5] [Online]. Available: https://github.com/PeterScott/murmur3

[6] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, “{SIEVE} is simpler than {LRU}: an efficient {Turn-Key} eviction algorithm for web caches”, in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1229–1246.