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

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. 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.

```c
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 [1] or similar light-tailed distribution. This means that there can potentially be performance gained from keeping common items in lower latency memory.

□

4. Results

5. Conclusion

Bibliography

[1] 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.