FreqTier: Lightweight Adaptive Tiering for CXL Memory Systems

Kevin Song[1], Jiacheng Yang[1], Zixuan Wang[2], Jishen Zhao[2], Sihang Liu[3], Gennady Pekhimenko[1,4]
University of Toronto[1] University of California San Diego[2] University of Waterloo[3] CentML[4]

1 Introduction and Motivation

Modern applications [2] demand increasingly large memory capacity and high bandwidth. However, main memory is already one of the most expensive components in data center servers. Meta reports that 37% of the total cost of ownership is spent on memory [6]. Moreover, the growth in DRAM density has been slowing down since the last decade [7], further limiting the capacity scalability of main memory.

Compute Express Link (CXL) memory tiering is a promising solution. Compared to local DRAM, CXL-attached memory has a larger capacity, but suffers from higher latency and lower bandwidth. Therefore, a tiering system should prioritize placing hot data in local DRAM (fast-tier) while keeping cold data in CXL memory (slow-tier) for better performance.

However, achieving high performance for a tiered system is challenging. We make two observations: 1) Real-world workloads often exhibit dynamically varying data hotness distributions [3]. For instance, hot data items can become cold in a matter of minutes [2], causing the hotness distribution to change dynamically. 2) Managing data access statistics can incur high overhead. Large memory servers with terabytes of memory capacity can contain billions of pages. Tiering metadata associated with each page combined can consume non-negligible memory thus decreasing tiering cost-effectiveness. In addition, frequently accessing the tiering metadata can generate high amounts of CPU cache traffic, causing resource contentions. Based on these challenges, an ideal tiering system should satisfy three requirements: 1) accurately capture the hot set by placing the hottest data in fast-tier memory 2) quickly adapt to changes in the hotness distribution 3) minimize tiering metadata overhead.

2 Prior Tiering Systems

We analyze prior works based on the above three requirements. One class of tiering system adopts frequency-based tiering [5], which stores the access count of each page to build an overall hotness histogram. To maintain freshness, such systems reduce all page access counts periodically, a process called "cooling". This cooling mechanism presents a tradeoff between requirements 1 and 2. A lower cooling period allows the tiering system to identify new hot/cold pages more quickly, since page access counters are refreshed more frequently. However, doing so also causes the hotness histogram to be less accurate, since it reduces the number of memory accesses reflected in the histogram. Furthermore,

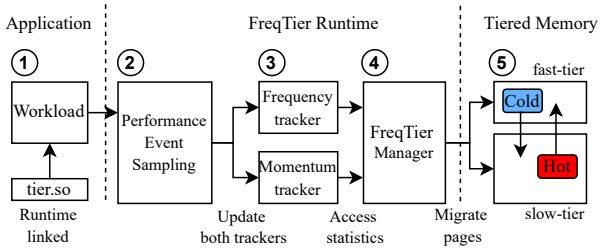

Figure 1. System overview of FreqTier.

to store access counts for billions of pages, prior frequency-based systems can incur gigabytes of memory overhead per server, translating to lower cost-effectiveness at the data center scale (requirement 3).

Another class of prior works [6] is recency-based tiering, which uses data access recency to approximate data hotness. Such systems do not consider a page's historical access statistics, but instead use recency metrics such as time between consecutive page faults to make data placement decisions. The main drawback of recency-based systems is their inability to accurately identify hot pages, since a recently accessed page may or may not be a hot page [5].

3 FreqTier Design

Adapting to Varying Hotness Distributions. Frequency-based tiering can effectively capture the overall hotness distribution, but cannot quickly adjust to changes. Recency-based tiering can identify new hot pages quickly, but cannot accurately capture the entire hot set. We observe that this tradeoff is the consequence of the fact that *prior systems only tracks one metric for each page*. Based on this observation, our key idea is to maintain *two* separate metrics for each page: "frequency" and "momentum". This is illustrated in Figure 1. Frequency tracks the number of accesses in the order of minutes to hours, while momentum monitors access intensity within seconds. FreqTier promotes pages with high frequency *OR* high momentum. This allows pages that recently became hot to be quickly promoted. FreqTier immediately demotes pages with low frequency *AND* low momentum. Pages with low frequency but high momentum are given a second chance, since such pages may be only cold temporarily. In practice, the frequency threshold is automatically adjusted based on the current hotness distribution (similar to Memtis [5]), while the momentum threshold is set empirically to 3. This flexible migration policy enables

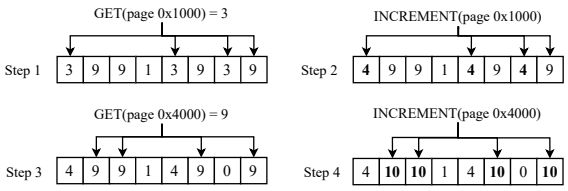

Figure 2. Counting bloom filter illustration. Counters represent page access counts.

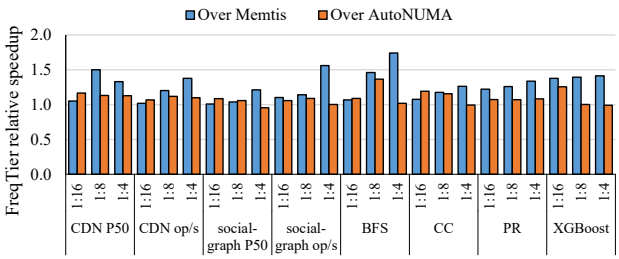

Figure 3. Regular page performance.

FreqTier to accurately capture the long-term hotness distribution (req. 1) while *simultaneously* quickly identify pages that are turning hot and turning cold (req. 2).

Metadata Memory Overhead. Prior frequency-based systems utilize *exact data structures*, such as hash table, to store page access counts. Exact data structures guarantee that a lookup will always return the previous latest value inserted. A hash table guarantees exactness by allocating dedicated memory for each item and resolving hash conflicts. We argue that exactness is not required for memory tiering. Intuitively, even if the access count of a very hot page is slightly inaccurate, it will most likely still be classified as a hot page. Following this intuition, FreqTier FreqTier adopts counting bloom filters (CBFs), a *probabilistic* data structure, to store page access counts. Unlike a hash table, CBF intentionally allow hash collisions to achieve higher memory efficiency. Figure 2 illustrates CBF operations. Our experiments show the inaccuracy due to hash collisions has a negligible impact on application performance.

Reducing Tiering Cache Overhead. The main source of cache overhead for prior tiering systems occurs during tiering metadata updates. For every memory access sample collected, the tiering system increments the access counter of the sampled page. Since the size of all tiering metadata can easily exceed the LLC cache size, frequent metadata updates can result in a large number of cache misses. To address this, FreqTier adopts blocked counting bloom filter [1], an optimization on top of CBF to guarantee each lookup will incur exactly one cache access and at most one cache miss.

4 Evaluation

Methodology. Similar to recent works [5], we use a two socket server to emulate CXL memory, where the local NUMA node is the fast-tier, and remote NUMA node the slow-tier. Each socket consists of a 16-core Intel Xeon 4314 processor and 512GB of DDR4 memory.

Baselines and Workloads. We compare FreqTier against Memtis [5] and AutoNUMA [4] on six large memory applications. CacheLib is a production-level in-memory caching engine [2]. GAP benchmark suite is a collection of standard graph processing kernel implementations. XGBoost is a widely used gradient-boosting library commonly executed on CPU systems.

End to end Performance. Figure 3 shows FreqTier performance on regular 4KB pages under various fast to slow-tier memory ratios (1:4 represents 128GB fast-tier and 512GB slow-tier). On average, FreqTier outperforms Memtis and AutoNUMA by 25% and 9% respectively. Compared to Memtis and AutoNUMA, FreqTier's performance improvements come from its better adaptability and lower cache overhead. Under huge page, FreqTier outperforms Memtis by 1.11× and 1.13× for 1:8 and 1:4 configurations respectively while performing on par for 1:16.

Metadata Overhead. In terms of tiering metadata size, FreqTier incurs 4.6× less metadata overhead than Memtis on average. This is because FreqTier uses more memory-efficient probabilistic data structures to track page access counts. In terms of cache overhead, compared to Memtis, FreqTier reduces the total number of L1 and LLC cache misses by 1.7× and 1.8× when using regular pages, and 3.2× and 3.5× under huge pages.

5 Conclusion

We propose FreqTier, an adaptive and lightweight tiered memory system. FreqTier quickly adapts to changing access distributions by tracking both long and short-term access statistics simultaneously. At the same time, FreqTier achieves low metadata memory and cache overhead by adopting probabilistic access frequency tracking.

References

[1] A high performance caching library for java. https://github.com/ben-manes/caffeine.

[2] Benjamin Berg et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX OSDI*, pages 753–768. USENIX Association, November 2020.

[3] Christina Giannoula et al. Daemon: Architectural support for efficient data movement in fully disaggregated systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(1), March 2023.

[4] Ying Huang. [patch -v4 0/3] memory tiering: hot page selection.

[5] Taehyung Lee et al. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th SOSP*, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Hasan Al Maruf et al. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM ASPLOS*, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Onur Mutlu. Memory scaling: A systems architecture perspective.