FrozenHot Cache: Rethinking Cache Management for Modern Hardware

Ziyue Qiu[†§‡] Juncheng Yang [‡] Juncheng Zhang[†] Cheng Li[†ζ] Xiaosong Ma[*]
Qi Chen[§] Mao Yang[§] Yinlong Xu[†ζ]

[†]University of Science and Technology of China [§]Microsoft Research
[ζ]Anhui Province Key Laboratory of High Performance Computing
[‡]Carnegie Mellon University [*]Qatar Computing Research Institute, HBKU

{ziyueqiu,juncheny}@andrew.cmu.edu,xzwj@mail.ustc.edu.cn,chengli7@ustc.edu.cn
xma@hbku.edu.qa,{cheqi,maoyang}@microsoft.com,ylxu@ustc.edu.cn

Abstract

Caching is crucial for accelerating data access, employed as a ubiquitous design in modern systems at many parts of computer systems. With increasing core count, and shrinking latency gap between cache and modern storage devices, hit-path scalability becomes increasingly critical. However, existing production in-memory caches often use list-based management with promotion on each cache hit, which requires extensive locking and poses a significant overhead for scaling beyond a few cores. Moreover, existing techniques improving for scalability either (1) only focus on the indexing structure and do not improve cache management scalability, or (2) sacrifice efficiency or miss-path scalability.

Inspired by highly skewed data popularity and short-term hotspot stability in cache workloads, we propose FrozenHot, a generic approach to improve the scalability of list-based caches. FrozenHot partitions the cache space into two parts: a frozen cache and a dynamic cache. The frozen cache serves requests for hot objects with minimal latency by eliminating promotion and locking, while the latter leverages the existing cache design to achieve workload adaptivity. We built FrozenHot as a library that can be easily integrated into existing systems. We demonstrate its performance by enabling FrozenHot in two production systems: HHVM and RocksDB using under 100 lines of code. Evaluated using production traces from MSR and Twitter, FrozenHot improves the throughput of three baseline cache algorithms by up to 551%. Compared to stock RocksDB, FrozenHot-enhanced RocksDB shows a higher throughput on all YCSB workloads with up to 90% increase, as well as reduced tail latency.

1 Introduction

Caching is one of the key techniques for accelerating data access and is deployed widely in systems across levels [51, 67, 72–74]. For I/O operations, its benefits stem from closing the significant performance gap between main memory and block storage devices, both in latency (up to 1000×) and bandwidth, by keeping the copies of hot items within and serving them from main memory [11, 19]. In addition, with today's trend toward resource disaggregation, large data centers begin to adopt a layer of cache servers [30, 59, 61, 68], assisted by common caching engines such as CacheLib [29] and Segcache [76], allowing diverse applications to benefit from the high throughput and stable latency offered by highly optimized memory caching.

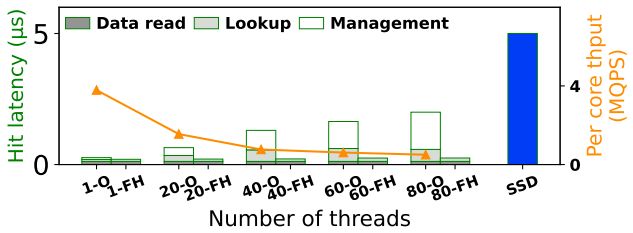


Figure 1. Hit latency breakdown for HHVM LRU cache implementations [12]: original (O) and FrozenHot (FH) with varied thread numbers. The line shows the decline in per-core throughput as concurrency grows.

As a transparent system optimization, the cache is known for its "live" adaptability achieved by dynamically admitting, promoting/aging, and evicting data items, collectively called cache management. However, cache management brings severe scalability issues and non-negligible hit latency for modern web applications.

Memory caches are further stressed as high hit ratios, and highly concurrent workloads are increasingly common in production systems [22, 46, 75]. Modern processors offer higher hardware parallelism, with over 100 cores per server and more to come in next-generation CPUs [14, 78]. Moreover, the deployment of fast storage devices further amplifies the impact of this software overhead.

Figure 1 shows the average hit latency of the Meta HHVM LRU cache [11] under a Zipfian workload (θ = 0.99), broken down into time spent on data read (4KB objects), cache lookup, and cache management. As the thread count increasing, the original HHVM implementation (O) on the left shows fast-growing latency. At 80 threads, due to lock contention, one cache hit takes more than 2 μs, 7.6× higher than at a single thread, over 70% of which comes from cache management. At this point, reading data from DRAM only accounts

for 6% of the latency. On the right side of the figure, we included the read latency of the Optane P5800 SSD used in our testbed, at around 5μs. With such cache contention, cache hits have lost their edge from in-memory data accesses to approach I/O speed. Accompanying such latency increase is the *per-core throughput drop*: we observe around an 80% decrease at 40 cores compared to a single core (Fig. 1).

Such cache-induced management creates inadvertent dark spins on even the most cache-friendly user access traffic: (1) originally independent user accesses to distinct data objects now experience heavy contention to shared cache data structures, and (2) originally read-heavy or even read-only user accesses now become write-intensive due to cache metadata updates. As a result, cache hits must undergo synchronization, forcing systems to spend much more time on cache management than on data access itself.

To achieve linear scalability, we present a new cache design, FrozenHot, which uses a new periodically rebuilt frozen cache to help reduce the management cost of the conventional dynamic live-updated cache. The FrozenHot design is based on the key observation that in a short period of time, the hottest items are relatively stable, as found in related studies [7, 10, 25, 32, 62]. We argue that the continuous management performed by conventional caches is often wasteful. Take the LRU cache as an example. When workloads possess strong temporal locality, many cycles are spent on rearranging items within the first half of the LRU list; when workloads have poor locality (such as scans), even more efforts are spent on replacement, without necessarily improving caching effectiveness (hit ratio).

FrozenHot virtually partitions the cache into two parts. The "frozen" part (FC) is expected to host the hottest items and serve them at extremely low latency as it removes cache management and related lock contention. The rest is managed as a conventional cache with dynamic insertion and eviction. Instead of being continuously updated, the FC is periodically reconstructed to adapt to workload changes. FrozenHot uses a lightweight background FC-controller to adaptively choose the frozen cache size based on access pattern, concurrency level, and cache size.

With its partially static design, FrozenHot reduces unnecessary management work and improves cache performance in multiple ways. First, unlike the common optimization to improve cache hit rate through cache algorithm designs [39, 41, 65], FrozenHot instead *completely eliminates the use of locks on the hottest items* by pinning them in the frozen cache without cache management for a sizable time interval, whose duration is automatically and dynamically configured. Accessing hot objects during these frozen times does not need locking, paying only lookup overhead on top of direct DRAM accesses. Second, such a frozen cache allows the adoption of more efficient data structures, such as fast, lock-free hash tables for quick and scalable indexing. These options are often beyond the reach of conventional caches

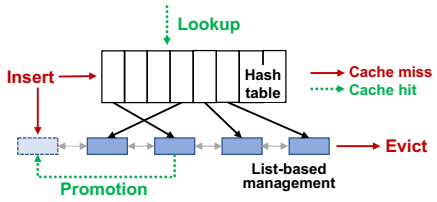


Figure 2. General architecture of list-based caches

accommodating replacement and resizing. Finally, for workloads with popular access patterns that are hostile to LRU/-FIFO (such as circular scans), FrozenHot could significantly improve the hit ratio by giving up dynamic management and instead holding on to a subset of data items.

The FrozenHot design is general and can be employed in many existing caches with minor code modifications. In this paper, we focus our discussion on the popular linked-list-based design (e.g., LRU, FIFO, 2Q [43], ARC [57], TinyLFU [36], etc.). We implemented a FrozenHot library and demonstrated its ease of use by enabling it in the HHVM LRU cache [11] and the RocksDB block cache [19], with under 100 lines of code change. Our design and evaluation illustrate FrozenHot's working with LRU, LFU, and FIFO.

Using production traces from Microsoft and Twitter, as well as synthetic Zipfian workloads, we show that on HHVM LRU cache, FrozenHot can improve throughput up to 551%. End-to-end evaluation using RocksDB shows that FrozenHot can improve its throughput by up to 90% and reduce tail latency by up to 46%. To the best of our knowledge, FrozenHot is the first caching scheme fundamentally satisfactory in both hit ratio and hit scalability.

2 Background and Motivation

2.1 In-memory Caching

As a critical building block for speeding up data access, in-memory caches are widely deployed to support today's online applications [29, 55, 75].

List-based management. We examined the designs of many widely used in-memory caches [4, 11, 15, 16, 29] and found that list-based management adopted *dominantly* in today's cache algorithm implementations. These include not only recency-based systems (such as ARC [57], LIRS [41], SLRU [45], and CACHEUS [65]), but also recent frequency-based approaches like TinyLFU [36] and LeCaR [69]. Therefore, this paper focuses on list-based cache implementations.

Fig. 2 shows the common design of prevailing list-based cache implementations today, with two main data structures: (1) an indexing structure (typically a hash table or B-tree) that facilitates fast data lookup and (2) a list structure that keeps track of the hotness of items accessed and facilitates cache evictions. Data access involves first looking up the index and obtaining a pointer to the requested data upon hits. This is followed by updating the list by promoting the fetched object, *e.g.*, moving its list element to the head with

LRU, or to the frequency-based location with LFU. When inserting an object upon a cache miss, both the indexing and list structures are updated. As a consequence, even with today's read-dominant application workloads, the cache internal operations all require updates protected by extensive locking, which hinders system scalability.

2.2 Trends in Modern Hardware

Modern storage systems have undergone a rapid transformation with a shrinking latency gap between storage hierarchies. At the top, large in-memory caches are routinely used, with potential expansion to larger, cheaper, and slightly slower persistent memory (PMEM) [13], such as Twitter's recent exploration [22]. At the bottom, backend storage often adopts faster NVMe SSDs [3, 31, 47]. Compared to the scenarios caches were designed to serve several decades ago, the much-reduced latency gap between the memory cache and storage layers implies that the cache management overhead becomes increasingly visible and could significantly erode the performance profit of caching.

In addition, running a cache on multiple cores in parallel requires synchronization on the indexing structure and list-based management: each request, hit or miss, needs to lock both the indexing structure and the list. This synchronization can significantly increase the latency of a cache hit, lowering the overall cache throughput consequently. We call this problem *hit-path scalability*, which occurs when the cache hit ratio and workload concurrency are high, and further exacerbated by fast storage backend. As we have seen in Fig. 1, scaling to 80 cores brings a 7.56× *hit latency* increase as compared with single-core execution, resulting in a significant per-core throughput reduction.

2.3 Existing Solutions to Improve Scalability

Lock-free data structure. Optimistic concurrency control over lock-free data structures has been shown to increase system throughput for read-dominated workloads [32, 38, 52]. However, it can only improve the scalability of the indexing structure, while we observe that *list management* is the main bottleneck for scaling in-memory caches. Therefore, existing lock-free data structures and techniques such as RCU [24, 56] cannot remove this major scalability hazard.

Trading efficiency for scalability. To reduce cache management overhead, several approaches exist to trade cache efficiency for scalability. For example, CLOCK [34, 38, 66] delays promotion until eviction time, HHVM relaxes LRU promotions using `try_lock` instead of `lock`, Cachelib [29] reduces promotion frequency, and Redis [18] uses sampling to evict an object this is *approximately* the least recently used. To achieve better scalability, these algorithms often sacrifice hit ratios, and sometimes bring other side effects. For instance, sampling objects to compare/evict in Redis [18] improves hit-path scalability. Meanwhile, it hurts miss-path scalability as one thread's sampled objects may be evicted by other threads, leading to many retries. Similarly, several recent cache eviction algorithms adopting sampling in promotion (*e.g.*, LHD [27] and LRB [67]) also suffer from this problem and have poor scalability.

Sharding and fine-grained locking. A common practice for better scalability is to partition the key space into shards, as adopted in production systems such as the HHVM LRU cache and RocksDB block cache used as baselines in this work [11, 16, 19]. Each shard maintains its own ordered cache list and indexing structure, mitigating global contention by per-shard locking. Sharding is orthogonal to and complements the two approaches described above. However, their combination still does not solve the list-management scalability problem, partially due to the obvious load imbalance side effect of sharding [28, 38, 40, 50, 53, 58]. Similarly to sharding, indexing structures also often adopt fine-grained locking to improve scalability [15], which, again, cannot help with list-based management.

2.4 Opportunities

Our approach is inspired by the following two observations.

First, many target workloads of in-memory caching today do possess strong locality brought by highly skewed popularity, where a small number of objects are accessed much more frequently than others, a natural behavior existing across applications from social networks to online shopping and web browsing. Companies like Twitter, Facebook, and Alibaba have confirmed that their cache workloads follow the Zipfian distribution with significant skew [25, 29, 32, 75]. Meanwhile, while these access hotspots gradually shift as time goes on, they remain fairly stable within short time periods [42, 49]. For example, recommendation systems often reuse cached results within a certain time window, periodically refreshing by incorporating new activities and content [76]. Timeline content display, pervasive in applications such as social networks, news feeds, and messages/emails, smoothly transitions user attention and traffic to newer content, while older objects get naturally aged when moved away from the front page. This combination of strong locality (high hit ratios) and short-term hotspot stability leads us to take a more radical approach targeting *removing cache management altogether for the majority of the average-case accesses (i.e., cache hits).*

Second, cache workloads (especially when caching storage blocks) could often exhibit scan or repeated scan access patterns [65, 69], which sometimes causes cache thrashing when the scan size is larger than the cache size and the cache uses a recency-based eviction algorithm such as LRU. Instead of improving hit ratio by trying to intelligently predict which content to keep in the cache when the workload contains a non-trivial amount of scans, our approach, in this case, may also decide to freeze a large fraction of the cache, significantly lowering the management cost during a period where shuffling cache contents leads to marginal profits.

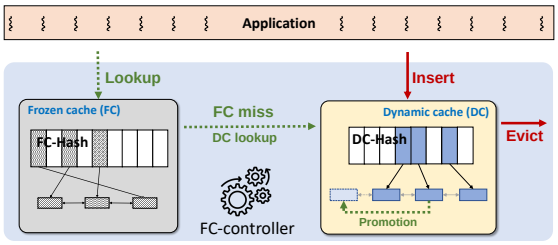


Figure 3. Overview of FrozenHot. A lookup first goes to FC-hash (FC); if it is a cache miss, then look up in DC-hash (DC). Insertions and evictions occur only in DC.

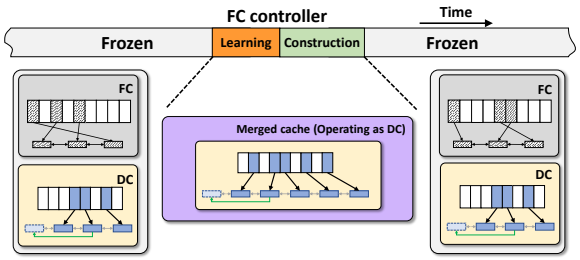


Figure 4. The three phases of FrozenHot

3 FrozenHot Design

3.1 FrozenHot Overview

FrozenHot works as a user-level library enhancing new or existing in-memory caches. FrozenHot builds on top of a base cache and has three components: a *frozen cache (FC)*, a *dynamic cache (DC)*, and a *FC-controller*. The FC is designed to achieve high throughput and linear thread scalability by freezing hot objects and *eliminating* locking. The DC runs with the original cache management mechanism, supplementing the FC. FC-controller is the brain of FrozenHot that decides how to build FC to provide optimal throughput.

Note that although FrozenHot divides the cache into FC and DC, it does not physically partition the cache space. Instead, it builds a separate "overlay" metadata structure (FC-hash) to identify objects in FC. Periodically, these two caches are merged for FrozenHot to learn about the current accesses, before they are split again to form the next FC, again through rebuilding the FC-hash. The data objects cached, on the other hand, are stored together and do not need to be migrated. The linked list is split between the two, with the FC segment not accessed and the DC segment used as usual with original cache management. Figure 3 depicts FrozenHot's overall architecture and access workflow. Below we give more details on its components and operations.

Frozen cache (FC). FrozenHot adds FC-hash, a fast, lock-free hash table that indexes to the FC-cached objects, allowing it to bypass the DC in their accesses. The FC-hash is periodically constructed to keep up with the current working set. Once set up, it remains immutable until the next reconstruction. The latency benefits of FC come from two sources. First, its read-only use of the FC removes the need for locking and hence avoids lock contention. This also reduces the number of required operations per request, trimming many bookkeeping tasks. Second, its read-only nature also enables the adoption of fast, immutable hash table implementation for the FC-hash.

Dynamic cache (DC). Working as the "base" cache, the DC has conventional management performing replacement based on a selected cache algorithm such as LRU and LFU. An FC miss results in a lookup in the DC, where a DC hit promotes the object within the DC according to the eviction algorithm. A DC miss requires fetching data from the backend and inserting data into DC.

FrozenHot operations. To serve a request, FrozenHot first checks the FC. If the requested key is found (*FC hit*), the requested data is read and returned to the user. Otherwise, FrozenHot looks up the DC, hoping for a *DC hit*. Both FC hits and DC hits are served from the cache; however, the lock-free, management-free FC hits are much faster. A cache miss happens when the key is not found in either cache, forcing the data to be fetched from the backend and inserted into the DC. Note that while the FC is static with its immutable metadata (cache replacement only occurring on the DC side), its cached data are not read-only.

FC-controller. The core of FrozenHot is the FC-controller (Fig. 4), which makes decisions on phases: (1) a *learning phase* for learning about the application accesses and configuring key FrozenHot internal parameters, (2) a *construction phase* for rebuilding the FC, and (3) a (typically much longer) *frozen phase* for fast and scalable cache service on top of the split FC and DC. When a FrozenHot-enabled cache is started, it only has a single "base cache", operating as the DC. At the end of the frozen phase, the FC and DC are merged into a single cache (again operating in the DC mode), from which we reconstruct the next FC. Fig. 4 illustrates this process and more details are given in §3.2.

FrozenHot performance. The mean access latency with FrozenHot can be roughly estimated using Equation 1, where H represents hit ratio and L represents latency. Because a DC cache hit occurs only upon an FC miss, its latency L_{DC} consists of the FC lookup and DC lookup. Similarly, L_{miss} includes both caches' lookup, in addition to the backend read, cache insertion, and eviction latency.

$$L = L_{FC} \cdot H_{FC} + L_{DC} \cdot H_{DC} + L_{miss} \cdot (1 - H_{FC} - H_{DC}) \quad (1)$$

Due to its lock-free, immutable design, FC hits are very cheap, with L_{FC} at a fraction of L_{DC}. With an effective FC that intercepts the majority of cache hits with workloads possessing good temporal locality, such fast FC hits occupy the bulk of data accesses (reasonably high H_{FC}) and become the significant contributor to FrozenHot's performance advantage.

Less obviously, FrozenHot also improves the DC hit performance: with many requests served by the FC, the DC-side lock contention is alleviated, resulting in lower L_{DC}.

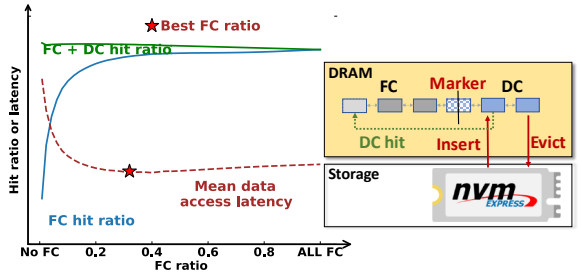


Figure 5. FrozenHot hit ratio curves and marker-based online profiling

FrozenHot's side effect is limited to the overall hit ratio ($H_{FC} + H_{DC}$) degradation: with the FC being static, it may not respond closely to changing working sets, resulting in a higher miss ratio. However, it adapts the relative FC size to the workload, and our experiments show that the overall hit ratio degradation is quite minor (< 1% in most cases).

Finally, one unexpected source of FrozenHot's performance gain is with scan-heavy workloads [65], which are unfriendly to mainstream cache algorithms. Here, FrozenHot can often be found freezing the entire cache when it realizes dynamic caching is not profitable. By doing this, it actually *improves the hit ratio* H_{FC} (by avoiding cache thrashing under cyclic scans) and simultaneously lowers the hit latency L_{FC} (by giving up the futile cache management altogether). Figure 11 will give related results.

3.2 Frozen Cache Learning and Construction

We now describe the strategies and mechanisms involved in periodic FC reconstruction, to answer the questions below:

1. How many and which objects should be frozen? (§3.2.1)
2. How to efficiently construct and deconstruct FC? (§3.2.2)
3. How often should the FC be reconstructed? (§3.2.3)

3.2.1 Learning Frozen Ratio. Freezing too few objects brings little benefits, while freezing too many could cause the hit ratio to drop significantly, increasing the overall access latency. Similarly, freezing for longer allows an FC to serve more requests after paying the reconstruction cost but comes with the risk of working under a deteriorated hit ratio. Therefore, the FC-controller learns these parameters from the workload through low-cost, online profiling and performs workload-aware auto-tuning to optimize overall cache service throughput.

To configure the FC size, FrozenHot defines `FC_ratio` as the ratio of FC size to FC + DC size. Unlike traditional cache sizes, here `FC_ratio` not only affects the cache hit ratio but directly impacts the hit latency, as revealed in our discussion following Equation 1. The left side of Fig. 5 illustrates the behavior of two hit ratio curves: one for FC and the other for the global cache (FC + DC). As we increase the `FC_ratio`, the FC hit ratio increases while the global one is likely to decline slowly due to the cache becoming more static. The sweet point (marked by the star in Fig. 5) yielding overall optimal throughput, corresponding to the lowest mean access latency, often sits somewhere in the middle.

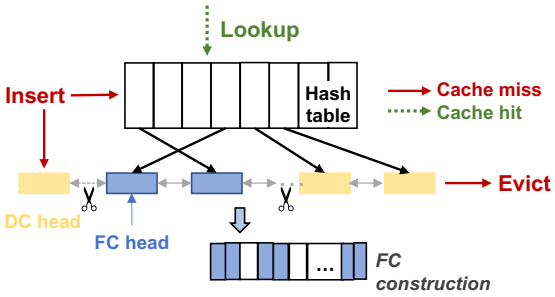


Figure 6. LRU-based FC construction.

In a learning phase, FrozenHot builds these curves using an efficient marker-based approach through the merged cache's normal operation as follows. We borrow the idea from zExpander [71], where a dummy element (marker) is inserted into the LRU list to monitor the time it takes to refresh the entire cache, *i.e.*, the time for this marker to be moved from the front to the end of the list. Here we use it mainly for a different purpose, to identify the desired location to split the LRU or LFU list. It inserts a marker at the head of the merged frozen list and doubly-linked list, and waits for the marker to traverse the list, as shown on the right side of Fig. 5. With the left end being the "most valuable" (such as the MRU and MFU end), the left side to the marker is viewed as FC, while the right as DC, allowing their separate hit accounting. Upon a DC hit, the object is promoted to the head of the list (in FC), and the marker moves down the list by one position. Meanwhile, cache insertions and evictions (caused by cache misses) are only allowed in the DC section, with newly admitted items inserted *to the right of the marker*.

During the marker's traversal, we count the number of objects promoted to the head of the doubly-linked list to estimate the current location of the marker. Although the workload may change during this process, we find such a simple learning scheme overall effective for a period longer than the marker's traversal. We also track caching statistics such as hit ratios and hit latency for the two (virtual) caches, as well as the overall miss latency. These data items are then fed into Equation 1 to plot the curves demonstrated in Figure 5, which then aid us in identifying the optimal `FC_ratio`. The FC-controller explores the space between 0% and 100% frozen with a step size of 2% (fully described in Appendix). To reduce profiling overhead, we adopt a sampling frequency of 1% (one data point for every 100 cache accesses), which is found sufficient in our experiments.

As the marker moves monotonically to the right while the merged cache serves requests, along the way we collect profiling data at selected `FC_ratio` stops, producing approximate FC and global hit ratio curves. Note that as such profiling allows the growing "FC segment" to operate without promotion, the learning phases themselves often outperform the baseline mode (by 19% on average).

3.2.2 Background (Re-)Construction.

Once the desired FC size has been learned, FrozenHot constructs the FC by first building the FC-hash, followed by splitting the cache list into two sub-lists: frozen list and doubly-linked list.

Fig. 6 gives an LRU-based example of the FC construction phase. We leveraged the insight that list-based cache management provides a convenient "top-k"-style interface to identify and separate FC objects quickly. For example, if FrozenHot selects a target `FC_ratio` of 20%, it freezes the left-most 20% of items of the merged list (in this case, the 20% most recently used objects). This is done by traversing the list and adding objects to the FC-hash till the target FC size is reached. To improve throughput, such construction is performed in the background. Meanwhile, since FrozenHot does not remove FC objects from the DC-hash, they are indexed by both hash tables, allowing the DC to serve all requests before the FC-hash is fully constructed. Deletion is handled by disabling the key in the FC-hash (enabled as an atomic operation) and marking it "deleted" in the doubly-linked list. Note that cache insertion and eviction take place in DC only, independent of the FC reconstruction.

Once the FC-hash is constructed, FrozenHot splits the cache list into a frozen list and a doubly-linked list in O(1) time complexity by pointer updates. During this short construction phase, promotion is temporarily disabled, and new objects are inserted at the head of the list (before splitting). Hence for LRU, splitting the frozen list requires cutting at two locations, as shown in Fig. 6. Among the three segments produced, the middle one is the frozen list, marked by a recorded FC head pointer. The list segment before the FC head and the segment after the `FC_ratio`-based traversing stop are then concatenated to form the doubly-linked list. Note that this would not happen with LFU, where inserts happen at the list tail, so a single list cut is sufficient.

After the list split, FC-hash concludes the construction phase and enters the frozen phase, during which the FC remains static and can be accessed without cache-management locks while the DC undergoes the standard list-based cache management. The split lists guarantee that FC-cached objects will not be evicted during the frozen phase.

Preparations for reconstruction. Upon the termination of a frozen phase, FrozenHot performs a two-step preparation: merging the FC and DC, and re-initializing learning. The former starts with first concatenating their lists recreating a unified cache list. FrozenHot then frees the FC hash table in the background. Because FC-cached objects are not evicted during the frozen phase and the FC-hash is a subset of the DC-hash, there is no need to copy/merge the hash tables. The latter requires FrozenHot to wait for the merged cache to warm up under the base algorithm (*i.e.* without FH). Here we determine the duration of such wait again with a marker-based approach, similar to that used in `FC_ratio` learning (Section 3.2.1). We insert a dummy object to the list head and trigger the learning phase upon the eviction, when the entire cache's content has been updated. Note that the learning phase is optional between two frozen phases, omitted if FrozenHot finds the learned `FC_ratio` in the previous episode works well (see results in Figure 13).

3.2.3 Adaptive Frozen Length.

FrozenHot builds its FC essentially by taking a snapshot of the items in the base cache that are considered most valuable by the baseline caching algorithm. However, this subset changes over time [29], often leading to the degradation of both the FC and the global hit ratios. It is also possible that access patterns may suddenly change, making a large fraction of FC content obsolete. Therefore, FrozenHot needs to adaptively decide the expiration time of the current FC during the frozen phase. To this end, we adopt three approaches, as described below.

Performance monitoring. During the frozen phase, FrozenHot continuously monitors metrics like average request latency and throughput. It terminates the frozen phase when performance deteriorates beyond baseline performance (obtained during the learning phase). In our prototype implementation, the FC-controller uses a simple window-based approach to monitor the average request latency as the criterion and exits the current frozen phase when the observed average drops below a configurable threshold which is based on the baseline. When FrozenHot terminates the current frozen phase due to performance degradation, it enters another learning phase where it re-learns the best `FC_ratio`, followed by a construction phase.

Periodic refresh. As a safety measure, FrozenHot also adopts a user-configurable *FC lifetime limit*, as a factor of the previous construct phase length (such as 20×). This forces FrozenHot to rebuild its FC periodically to protect it against being trapped by an unusually low baseline performance that happened to be captured during the previous learning phase. However, with such forced periodic refresh, the learned `FC_ratio` is still valid, unlike in the case of performance-based reconstruction (triggered by the aforementioned monitoring), so FrozenHot skips learning and directly enters construction. In other words, in this case, it rebuilds the FC without searching for the best `FC_ratio`. During the construction, however, it does monitor the baseline performance as a reference for performance monitoring in the next frozen phase.

Regression to baseline. Under challenging workloads such as those with highly dynamic behaviors, FrozenHot reserves safety measures to return to the baseline cache management (no FC). When it finds itself unable to beat the baseline performance during a learning phase, it starts a "waiting interval" for the cache to run in its baseline mode before it tests the water again. The waiting interval grows exponentially with consecutive unsuccessful trials.

3.3 FrozenHot for All

FrozenHot is designed to support all list-based cache management algorithms. The core idea of FrozenHot remains the same: freezing a subset of the most popular objects so that most cache hits can be served without locking. This strategy is based on the common "top-k" property mentioned earlier of list-based cache management, regardless of the underlying list-sorting metrics. Aside from the list split maneuver discussed earlier, the difference between major algorithms is mainly reflected in how we classify whether a cache hit is an FC or DC hit when serving from a merged cache for our low-cost online profiling during the learning phase.

Recency-based algorithms Our previous discussion showed a recency-based example, and the majority of our performance evaluation is with such implementation, as this is the dominant strategy adopted by current systems, found in many cache eviction algorithm designs such as FIFO, LRU, ARC [57], SLRU [45], LRU-K [63], TinyLFU [36], 2Q [43]. With these algorithms, we again utilize the marker, which naturally moves from the head to the tail.

More specifically, each DC list item is augmented with a timestamp recording its insertion/promotion time during the learning phase. With the marker logically partitioning the FC and DC segments, for each data hit, we could easily judge which segment it hits by comparing its timestamp with the marker's. If it is newer, we count this as an FC hit, otherwise, a DC hit. Only in the latter case the marker shifts one step toward the tail. This way, we easily collect profiling data points for the FC/global hit ratio curves with a growing `FC_ratio`, without repeated list traversal.

Frequency-based algorithms Another major class of cache management algorithms adopted in systems today is frequency-based, with LFU variants such as window LFU [44], LeCaR [69], and CACHEUS [65]. The design of FrozenHot for list-based LFU [54] is mostly the same as with LRU, except that the marker approach cannot be used here. If we insert a dummy marker, unlike with LRU, it will not naturally traverse the list but will likely stay at the tail and get evicted quickly. On the other hand, the LFU list needs to be strictly sorted by frequency, which allows us to traverse the list with a pointer and use the frequency value at its position as the *frequency threshold* for classifying FC and DC hits (accessing an item with the same or higher frequency counted as an FC hit). By keeping track of the pointer movement, again, we can easily profile the hit ratio curves at different `FC_ratio`, all during the merged cache's baseline operation.

Another difference is with the FC-DC merge at the end of a frozen phase. Unlike LRU, where we simply concatenate the two lists, here we need to ensure the merged list is sorted in frequency, while the FC list skips frequency maintenance when frozen. For better performance, FrozenHot performs a simple approximation by joining the lists first, then adding in the background a "frequency boost" equal to the DC list head frequency value at the time of the list merge.

```
1  void ConcurrentLRUCache() {
2    ...
3    FH = new fhcache(m_head, m_tail, m_map,
           m_listMutex,
           null); //FrozenHot cache initialization
4    ...
5  }
6  Obj* find(Obj* key) {//changes in find
7    if(FH.on) {
8      v = FH.FC_search(key);
9      if(!v){ //not FC found
10       v = m_map.search(key); //DC search
11       if(v){
12         if(!FH.constructing) //skip if FC
               constructing
13           ... //DC LRU update
14       }
15     }
16   }
17 }
18 void insert(Obj* key, Obj* value) {//changes in insert
19   if(FH.allfrozen)
20     return; //skip the DC LRU update
21   ...
22 }
```

Figure 7. Adding FrozenHot to HHVM LRU Cache, with lines in bold highlighting FrozenHot related changes

Note that FrozenHot's design is not specific to the metric used for list sorting. When the frequency is replaced by a custom-defined priority rating (such as GD [77], GDSF [81], and GD-wheel [48]), the same handling applies, and FrozenHot can be easily enabled.

4 FrozenHot Deployment

We now discuss the interfaces FrozenHot exposes to the cache developers, followed by case studies of enabling it in two production cache systems with minor code changes.

4.1 FrozenHot Library

FrozenHot augments existing cache design by adding additional metadata to "annotate" the frozen cache. Meanwhile, the dynamic cache reuses the cache design and implementation in the baseline system. We built FrozenHot as a lightweight user-level library in under 1000 lines of C++ code. The implementation consists of two major components, the FC-controller (to run as a daemon thread) that adaptively learns the `FC_ratio` and monitors FC performance for reconstruction, and a lock-free hash table as the FC-hash.

FrozenHot targets low-cost online decision-making with low overhead. First, as mentioned earlier, it does not require additional profiling, but piggybacks its online profiling on the routine operations of the merged cache, using coarse-granularity sampling and high-level accounting. Second, FrozenHot adopts existing state-of-the-art high-performance design and implementations. For example, it chooses CLHT [8], to implement the immutable FC-hash. As to space overhead, the major additional data structure is the FC-hash itself, plus very limited additional metadata such as the timestamp for LRU list elements. With 4KB-sized objects, for example, its space overhead will be no more than 0.6%.

4.2 Enabling FrozenHot in Production Systems

The FrozenHot library can be easily integrated into existing caches that use list-based management, covering the vast majority of production in-memory caches. To our best knowledge, the only exception is Redis [18], which we included in our performance evaluation. In this section, we report our experience in supporting FrozenHot in HHVM and RocksDB. Such adaptation requires under 100 lines of code change for either system.

HHVM [11] is an open-source virtual machine designed for executing programs written in Hack [1], with a built-in cache implementation to speed up data access. The HHVM cache uses a relaxed LRU for eviction and adopts the Intel TBB (Threading Building Blocks) library for high-performance concurrent hash table implementations [64]. The relaxed LRU uses list-based management as described in §2.1, as well as a mutex-based synchronization to coordinate concurrent accesses such as insertion, eviction, and promotion. To reduce lock contention, HHVM uses best-effort locking and promotion: instead of acquiring a conventional lock for object promotion on each request, it uses `try_lock`: if a thread cannot obtain the lock, it will not promote the object. In the extreme case, HHVM LRU cache becomes FIFO.

Figure 7 outlines the changes made to the HHVM cache at three locations. First, during initialization, cache developers inform FrozenHot about the major cache data structures and start the FH-daemon, which runs the FC-controller after initialization. This is done by creating the `fhcache` object (line 3), passing the LRU list head, list tail, and the hash table (`m_map`), as well as optional locks associated with them. Here no lock is passed for the hash table, as the TBB-based `m_map` comes with lock protection. The LRU list and hash table passed from the base cache implementation will be used as the FrozenHot DC list and DC-hash.

Second, cache developers need to add FC lookup to the find function, as shown in line 7-16. When the flag `FH.on` is set, indicating the status of being within a frozen phase, cache accesses need to first check the FC Hot-Hash, bypassing the DC if it is a hit (line 7-9). Additionally, a conditional check is needed to temporarily bypass LRU promotion if there is ongoing background FC construction, since we do not know which list (FC or DC) the element belongs to. (line 12-13).

Finally, in the cache insertion function, developers need to add a check that if FrozenHot decides to freeze the entire cache (indicated by the `FH.allfrozen` flag), the insertion is skipped upon DC miss (line 19-21).

RocksDB, a leading KV store deployed in many production systems [5, 21], uses a DRAM-based block cache [20]. Enabling it to use FrozenHot requires quite similar modifications as described above for HHVM. A major difference is due to RocksDB's log-structured feature: cache invalidation could be generated by compaction, where an explicit `Erase()` function is used for this purpose. Thus, we have a corresponding adaptation for this interface. Like with HHVM, the FC checks for deleted objects during the frozen phases.

5 Evaluation

5.1 Experimental Setup

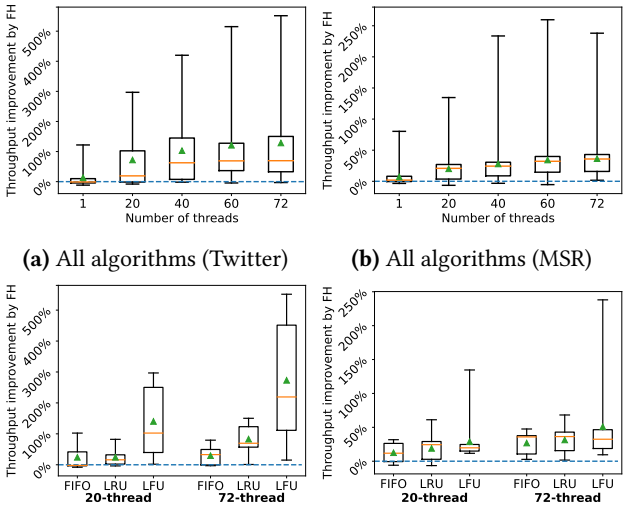
Hardware configuration. Our tests use two dual-socket servers with Intel Xeon (R) Platinum 8360Y 36-core processors (running Ubuntu 18.04.1) and 8380 40-core processors (running CentOS 8.5). Both have 256GB DRAM and almost identical per-core processing capacity. The former is used for experiments with traces while the latter, with a 1.5TB Optane P5800X SSD for storage, is used for RocksDB runs. To avoid NUMA impacts, our evaluations only use a single socket with hyperthreading enabled.

Baseline and system configurations. We used multiple cache implementations as baselines: the classic LRU ("LRU"), list-based LFU [54] ("LFU"), the optimized LRU used in HHVM that adopts `try_lock` to ease lock contention ("Relaxed-LRU"), FIFO ("FIFO"), and the optimized LRU used in Redis [17] ("Sampling"), which uses sampling to choose candidates for eviction. We built FrozenHot-enabled algorithms as described in §4 and call them "LRU-FH", "LFU-FH", and "FIFO-FH", respectively.

In microbenchmarks, to isolate File-System block caching related performance behavior, we emulated disk accesses for cache misses, adopting a 5μs latency based on our profiling results on the Intel Optane P5800X raw device. We used up to 72 client threads to generate workloads.

We also evaluated FrozenHot-enabled RocksDB for an end-to-end system performance study on a production LRU cache. We used four 1GB MemTables, following recent studies [31, 80] in adopting larger MemTables than the default setting for better baseline performance. Because RocksDB performs actual disk accesses, this end-to-end evaluation result demonstrates the user-perceived performance gain in a real-world deployment. We used up to 64 threads for RocksDB due to its having background threads for internal operations such as compaction and flush.

Workloads and datasets. We evaluated FrozenHot with a variety of workloads. To start with, we used two collections



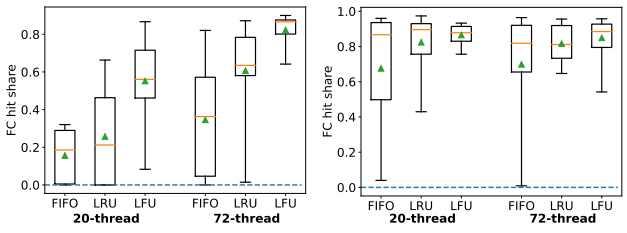
(a) All algorithms (Twitter) **(b)** All algorithms (MSR)

(c) By algorithm (Twitter) **(d)** By algorithm (MSR)

Figure 8. Throughput improvement for all algorithm-workload combinations, with increasing thread counts. Box shows the 25^{th} and 75^{th} percentile distribution, with whiskers extending to the min/max. The red line shows the median speedup while the green triangle the mean. The blue dotted line indicates the baseline.

of traces with contrasting characteristics. The first includes seven Twitter traces [2, 75], namely cluster 17, 18, 24, 29, 44, 45, and 52. These traces possess mild hotspot migration and are used in recent cache studies [55, 76]. The second consists of all twelve MSR Cambridge traces [23, 60] that have over 5M I/Os, which have been noted to possess a hybrid access behavior, with Zipfian-like accesses mixed with large scans [27]. Unless otherwise noted, the FC lifetime factor is set at 20× by default. In addition, for sensitivity studies we used synthetic workloads with the Zipfian parameter $\theta = 0.99$, representative of workloads from e-Commerce websites [32] and social networks [75].

For RocksDB, we ran the popular YCSB benchmarks [33], using a 100 GB database with 1 KB KV items. We measured performance after a cache warm-up phase, with FrozenHot operating on its base cache (DC).



(a) By algorithm (Twitter) **(b)** By algorithm (MSR)

Figure 9. FC hit share with FH versions of algorithms

5.2 Overall Performance: Trace-driven evaluation

We first report the overall performance of FrozenHot with the three popular caching algorithms (FIFO, LRU, and LFU) over the aforementioned 7 Twitter and 12 MSR traces. Cache size settings are workload-dependent. With Twitter traces, we adjust the cache sizes to re-produce the overall LRU hit ratio (without FrozenHot) as reported from production settings [6] (see Table 1). With MSR traces, we set the cache sizes at 10% of the dataset size, to provide low-locality test cases contrasting with the Twitter set. These per-trace cache size settings are fixed for all experiments. The FC lifetime factor is set at 20× (focused study in Figure 14).

Table 1. Cache size settings in this work.

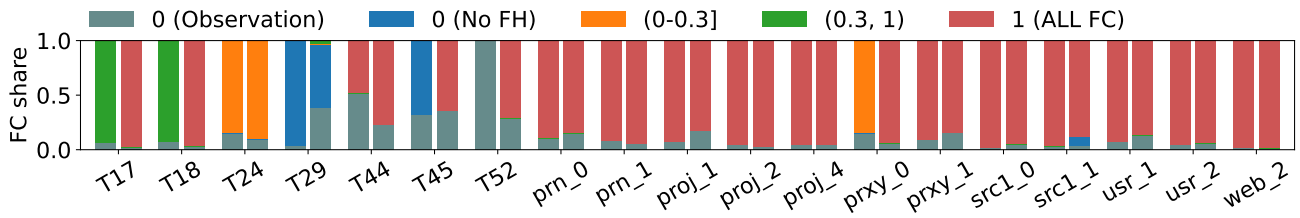
Twitter Trace	Cache Size (million)	# Object (million)	Proportion	Hit Ratio
Cluster17	1	6.1	16.4%	99%
Cluster18	30	45.4	66.1%	99%
Cluster24	2.4	11.4	21.0%	99%
Cluster29	2	11.9	16.7%	78%
Cluster44	40	57.7	69.3%	99%
Cluster45	88	243.2	36.2%	63%
Cluster52	0.9	239.2	0.4%	99%

Due to space limitations, Figures 8a and 8b summarize the throughput improvement brought by the FH implementation over the baseline algorithm, for all algorithm-trace combinations, under growing concurrency levels.

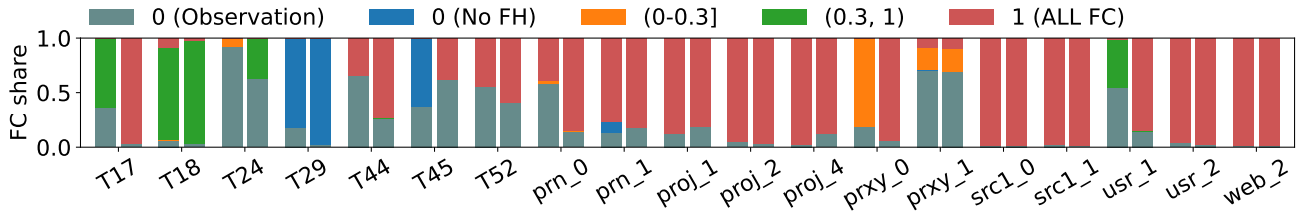
With no concurrency (1 thread), in most cases, FH works similarly to baselines. Interestingly, however, certain traces (*e.g.*, Twitter-52) also observe performance acceleration on LRU and LFU, with faster cache hits and nearly no miss ratio degradation. With increasing concurrency, FH's improvement steadily grows for both trace collections. At 72 threads, the FH versions offer a throughput improvement of up to 79%-551% over their respective baselines across workloads. Understandably, the Twitter collection sees more enhancement due to its much higher hit ratio (average at 90% as opposed to 37% with MSR). Still, even the much less cache-friendly MSR traces receive an average throughput boost of 36% with such high concurrency. The Twitter traces, meanwhile, have an average improvement of 118%.

Next, we zoom into the per-algorithm results in Figures 8c and 8d, here at two concurrency levels (20- and 72-thread runs).[1] Regardless of the workload or concurrency level, overall LFU benefits the most from FH, while FIFO the least. This is intuitive considering the hit-path overhead of the

[1]Though not included in these results, we did evaluate sampling-based LRU (the aforementioned "Sampling") here, and found it inferior to FIFO-FH (also trading hit ratio for lighter management). On all Twitter+MSR traces tested, FIFO-FH outperforms "Sampling" by 36% (20-thread) and 69% (72-thread) on average. Figure 13 and Figure 16 give more results involving "Sampling".



(a) LRU Algorithm



(b) FIFO Algorithm

Figure 10. This clustered, stacked bar graph shows the portiton of the runtime of different FC ratios when running different traces with the LRU and FIFO algorithms under two concurrency levels. The left and right bars in each cluster/pair represent settings with 20 and 72 threads, respectively. The observation period operates under the base algorithm (hence with an FC ratio of 0) is colored gray to distinguish it from a FrozenHot decision not to freeze.

three algorithms, with LFU at the top and FIFO at the bottom (see details in Fig. 12). The high-maintenance LFU is hence more workload-sensitive, bringing a much larger span of FH-induced improvement for Twitter. However, all algorithm-workload combinations benefit from FH, again with larger gain under higher concurrency. Absolute performance across algorithms will be discussed in Figure 14.

Also, we note from Figure 8 that in the worst case, FrozenHot generates little negative impact on cache performance. The few workloads that see a slight slowdown are from traces yielding very low hit ratios (the worst being MSR proj_2, at 6%) and running at low concurrency, where FH decides to have a very small or no FC after a few dis-encouraging trials.

Figure 9 has the same structure as 8c/8d, but a different metric showing FH's working with different algorithms. We show the overall *FC hit share*, defined as "FC hits divided by total cache hits".

As expected, both workloads show FC carrying more hits as concurrency increases. The inter-algorithm difference, meanwhile, confirms the positive correlation between the effective FC size and the cache management cost (LFU>LRU>FIFO). The large span of FC hit share (especially in Twitter cases like LRU-FH at 20-thread and FIFO-FH at 72-thread) shows that FH faces a wider range of workload- and algorithm-dependent choices, where the FC controller is smart enough to configure the FC size adaptively.

To zoom in on different `FC_ratio` options, Figure 10 further presents the distribution of `FC_ratio` selections. We observe that FC-controller tends to favor higher `FC_ratio` values under conditions of high contention, as evidenced

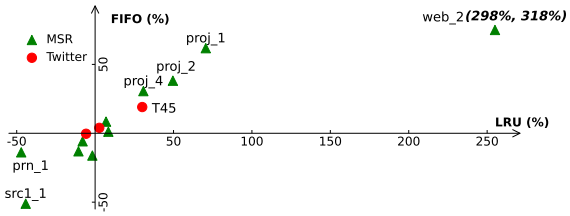


Figure 11. Relative change to hit ratio brought by FH. The green triangles and red dots show data points of MSR traces, and Twitter traces, respectively.

by the comparison between LRU (10a) and FIFO (10b), as well as the comparison between the 20-thread (left) and 72-thread (right) runs of each algorithm-trace combination. The left-most several Twitter traces do consume longer learning periods due to their more dynamic nature, yet still often produce a fully frozen cache and receive performance benefits from it, as seen from Figure 8 and results below.

Finally, Figure 11 shows the relative hit ratio change brought by FH for LRU (x-axis) and FIFO (y-axis), for the 19 traces tested. For better display, we omit five traces where the change is under 2% *for both algorithms*. One sees that the majority of the remaining 14 data points receive from FH a *hit ratio boost* despite being partially static. As discussed in §3.1, this is due to thrashing reduction for scan-rich workloads, which for web_2 brings a 300% hit ratio growth (from 3.5% to 15%). For a trace with moderate locality (Twitter 45), FH improves its FIFO hit ratio from 59% to 70%, and LRU from 61% to 80%.

Among the traces seeing lower hit ratios, prn_1 and src1_1 stand out. They present an interesting case where FH helps

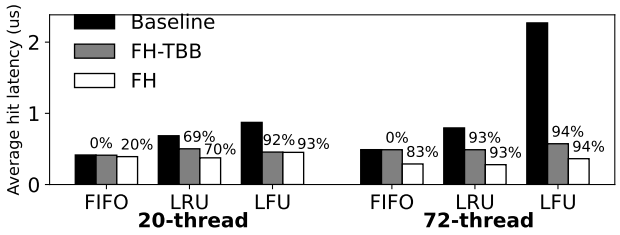


Figure 12. Average hit latency with Twitter cluster17 trace (showing representative behavior), with FC hit share annotated on top of the FH bars

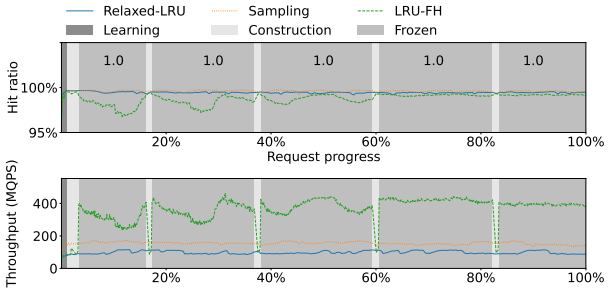


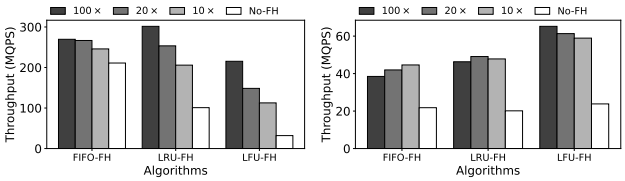
Figure 13. Sample FH phases (Twitter cluster17, 72-thread). Numbers inside show each frozen phase's `FC_ratio`.

in a surprising manner: with such cache-unfriendly traces (base hit ratio under 44% and 25%, respectively), FH decides to freeze the whole cache, so it basically optimizes the *miss latency* instead, by removing cache management. Therefore, these two traces receive an overall LRU/FIFO throughput improvement of 1%/10% and 15%/10%, respectively. Note that this benefit also applies to low-locality traces that receives a hit ratio boost, such as web_2 discussed above.

5.3 Sources of Optimization

FrozenHot's performance gain mainly comes from two major optimizations for FC hits: (1) removing management (especially lock operations) and (2) using a fast, read-only hash table (FC-hash). To assess their relative impact, we implement an intermediate version, "FH-TBB", with the first optimization only (frozen cache, but using the baseline TBB hash table implementation for both FC and DC). Figure 12 plots the average hit latency of the baseline, FH-TBB, and full FH, for each algorithm. For the two FH variants, we give the FC hit share on top of the bars again.

These results reveal that the largest gain comes from removing the list-based management for LRU and LFU, especially at higher concurrency. At 72 threads, FH-TBB brings a 39% hit latency cut for LRU and a dramatic 75% for the more costly LFU. As expected, FIFO, on the other hand, sees no improvement at all with FH-TBB (which decides to turn off FC, producing a 0% FC hit share). This is due to its very low management cost, as no promotion is needed for hits. Its improvement, therefore, comes purely from adopting a faster read-only hash table, which leads FIFO-FH to adopt



(a) Twitter cluster17 **(b)** Twitter cluster45

Figure 14. FH sensitivity to FC lifetime factor

FC sizes that take a 20% share in the 20-thread case, and 83% in the 72-thread (where FH cuts the hit latency by half).

This shows that even the "simple and cheap" FIFO algorithm could benefit significantly from FH, as shown in the hit latency cut, in high-concurrency scenarios. Meanwhile, the much larger magnitude of optimization FH brought to the smarter yet more costly LRU and LFU enable them to become competitive again. For example, with 72 threads, LRU-FH sees a similar hit latency as FIFO-FH, while producing a slightly higher hit ratio, allowing it to deliver an overall throughput 20% higher (more analysis on the heightened throughput sensitivity to hit ratio with FH to be given next).

5.4 FrozenHot Dynamics

We further zoom into the three phases of the above sampled execution (Twitter cluster17 with 72 threads). Figure 13 portrays the changes in the global hit ratio (top figure, starting from 95%, annotated with `FC_ratio` numbers) and throughput (bottom figure). We add "Relaxed-LRU" and "Sampling" as references, aligned to LRU-FH's request progress along the x axis, though they do not have the phases.

For this particular case, FH chooses to freeze the entire cache, perceiving that reducing hit latency outweighs partially losing adaptivity. Note that reconstruction activities were triggered either by the frozen period's expiration or observed performance degradation over a given threshold. Here we only had the first case, so learning was not activated except for the first FH episode.

As expected, while the other two solutions bring quite stable and consistent hit ratios (with the "Relaxed-LRU" and "Sampling" overlapping well), LRU-FH sees a gentle yet steady decline in hit ratio, especially within the 1st, 2nd, and 3rd frozen episodes. One may perceive such decay as quite minor, from 99% to 96%, but from the throughput results, such a decline brings a sizable performance drop. The reason behind this is that with the frozen cache, the gap between a hit (most likely an FC hit) and a miss becomes much more dramatic than experienced by the baseline solutions, confirming the necessity of periodic FC reconstruction with real workloads possessing hotspot migration.

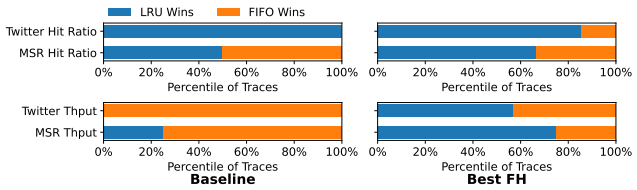


Figure 15. LRU vs. FIFO, before and after enabling FH

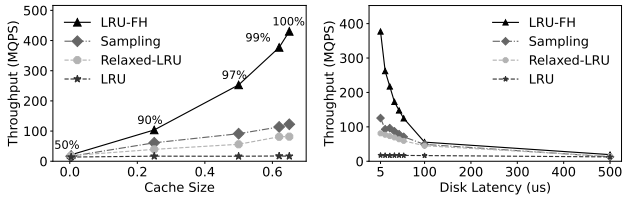
5.5 Sensitivity Analysis

One important FH internal parameter is the FC lifetime, after which an FC reconstruction is forced even without observed performance degradation. Figure 14 shows FH's sensitivity to it by varying the FC lifetime factor (100×, 20×, and 10×, giving growing FC refresh frequency) in 72-thread runs. The last bar ("No-FH") is included as a reference, with FrozenHot turned off. In addition to Twitter cluster17, here we add cluster45, which possess a much weaker locality.

With its mild hotspot migration, cluster17 performs best with infrequent FC rebuilding (100×). Cluster45, in contrast, has faster hotspot migration and benefits from more frequent FC refresh, with lifetime factors 10× and 20× winning for FIFO-FH and LRU-FH, respectively. Interestingly, cluster45 demonstrates LFU-FH's potential in production usage, where it outperforms the other algorithms by delivering higher hit ratios (85% vs. FIFO-FH's 70% and LRU-FH's 80%), when its heavy management overhead is alleviated by FH.

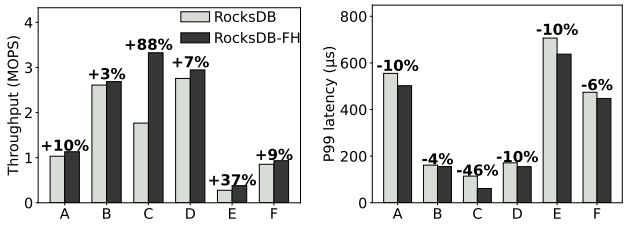
These results also confirm that FrozenHot can keep its FC frozen for much longer than the construction phases. While not surprisingly, FC lifetime setting is workload-dependent, this study shows that a fixed, moderate setting (like 20×) works fairly well across the board and yields significant gains against the "No-FH" baseline. Though workload-adaptive FC lifetime selection is left for future work, below we extend our study to examine the three popular caching algorithms' relative strength with FH's best lifetime limit setting.

Figure 15 summarizes FH's impact on the FIFO-LRU face-off, measured by calculating the share of trace workloads each policy "wins", among the Twitter and MSR traces tested. The left side gives baseline results. While LRU excels in hit ratio (claiming all Twitter and half MSR traces), FIFO shines in overall performance (winning all Twitter and nearly 80% of MSR traces). This echoes an earlier study [37], which finds FIFO competitive by its simplicity. On the right side, we show how FH changes the game, with the best FC lifetime settings (100×, 20×, or 10×) found in the previous set of tests. Interestingly, with FH, LRU loses in the hit ratio fight in Twitter traces (less adaptivity), but gains in MSR (less churning). However, for both trace sets, by trimming cache management overhead, LRU-FH emerges as the overall winner, beating FIFO-FH in throughput with 68% of the traces.



(a) Cache size impact **(b)** Storage device latency impact

Figure 16. Sensitivity analysis with Zipf 0.99 at 72 threads. Cache size is the fraction of dataset size.



(a) Throughput **(b)** P99 latency

Figure 17. Overall YCSB performance (Zipfian 0.99), with improvement or reduction ratio annotated at the top

Finally, we conduct a set of tests on cache size sensitivity with a synthetic Zipfian 0.99 workload. In Figure 16a, we adjust the cache size relative to the total data footprint. The results demonstrate FrozenHot's advantage in sustaining its effective utilization of additional cache size, even achieving a faster throughput growth when the cache size grows from 60% to 80% (corresponding to hit ratio from 99% to nearly 100%), as the LRU-FH hit latency is a very small fraction of the miss latency. The baseline systems, on the other hand, suffer from their success as the gain from the higher hit ratio is lost in severe lock contention. Note that the sampling approach helps little here, as its hash table (protected by read-write locks) is also costly with high hit intensity.

Figure 16b studies the impact of varying latency of the underlying storage layer on different solutions. As expected, FrozenHot is able to take advantage of lowering device latency (moving to the left along the x axis), as it is designed to handle the increasing hit intensity (brought by faster misses), as well as to tolerate a higher miss ratio with storage devices becoming faster. The best baseline approach ("Sampling"), meanwhile, is able to achieve the throughput that FrozenHot delivers with a device latency of 35us, on an ultra-fast device with latency at 5us.

5.6 RocksDB End-to-end Performance

We report RocksDB results running YCSB benchmarks in Figure 17. We used a 100GB database, with 30GB memory cache size. Performance measurement starts after cache warm-up, with each test issuing 20M requests. Here we report its overall performance, where the end-to-end request processing

contains much more tasks beyond caching. Still, FrozenHot can generate a considerable throughput improvement and reduce tail latency for all workloads.

The read-only YCSB-C sees the most gain from FrozenHot (throughput improvement of 88%), as it benefits the most from the in-memory cache and could enjoy FH's low hit latency. For three other workloads with 5% writes, YCSB-B, D, and E, FH brings smaller improvement, 3%, 7%, and 37% respectively, due to its enhancement of read hits. In particular, it is able to handle well the scan-heavy operations in E. In addition, the better baseline performance with B and D comes from MemTable hits brought by the 5% writes.

Two write-heavy workloads here, YCSB-A, and YCSB-F contain 50% reads and 50% writes, update, and read-modify-write respectively. Both workloads see lower baseline performance, while FrozenHot is able to bring around 10% throughput enhancement by speeding up reads.

Figure 17b gives the P99 latency. The results show that FH is able to bring tail latency reduction across the board, in addition to lower average latency (corresponding to the higher throughput), mainly due to reduced lock contention.

6 Other Related Work

Maximizing hit rate. A large body of recent cache management designs focus on optimizing the cache hit ratio [39, 41, 65, 71]. zExpander compresses cold blocks to cache more items in memory [71]. HashCache [26] uses less memory without losing performance, allowing it to share resource with other tenants. Segcache addresses cache inefficiencies caused by large per-object metadata and clumsy item expiration [76] via object metadata sharing and bulk expiration and eviction. Similarly, MemC3 [38] improves memory efficiency by reducing per-object metadata using CLOCK and a compact hash table. Most recently, CACHEUS [65] and its ancestors have applied machine-learning techniques to identify suitable cache replacement policies for various workloads. HARC[39] is a hierarchical adaptive cache replacement policy that takes into account the dirty, clean, recency and frequency of cached memory pages. The idea of FrozenHot is orthogonal to this set of works.

Reducing cache management overhead. MemC3 [38] additionally adopts optimistic locking with the limitation of supporting multiple readers but a single writer. Some recent approaches suggest using simpler replacement policies than LRU, such as CLOCK and FIFO, to trade cache hit ratio for lower management cost [37, 38]. Segcache [76] adopts a "macro-management strategy" to replace per-request bookkeeping with batched operations on segments to reduce locking overhead. Furthermore, sharding has been widely used in production [11, 16, 19], for example, RocksDB uses 16 shards, which improves scalability but does not reduce management overhead. Though these optimizations trim down the lock contention to some extent, the reduction still cannot meet the ever-evolving hardware advancements with denser CPU cores and faster disk drives. Unlike these proposals, FrozenHot stands from a different point of view to propose a new cache management scheme that chooses not to pay unnecessary management cost for hot objects.

Optimizations for fast devices. The arrival of fast storage media invalidates the assumptions on which conventional cache designs are based and thus inspires new innovations. For instance, NHC [70] maximizes bandwidth usage of both the caching device and the capacity devices by dynamically moving excess load to the capacity layer. FrozenHot focuses on improving the performance of the cache layer, which is complementary to NHC. Eytan et al. conducted a study on comparing the performance of LRU and FIFO cache replacement policies in the new cloud environments where the storage can be very fast, and shared the similar sentiment that the cache hit ratio no longer dominates the results when the performance gap between the cache and backend storage gets smaller [37]. Compared to these works, FrozenHot is a general approach for improving the hit-path scalability of a list-based cache. As the memory/storage hierarchy becomes more complex, some recent approaches explored data placement across hierarchy [35, 79, 82, 83]. However, they often have a higher management cost and lower throughput. Therefore, we focus on caching and improving its management cost under high concurrency.

7 Conclusion

In this work, we reflect on the common practice of retaining valuable data using constant cache management. We argue that with today's fast I/O devices and highly concurrent execution environments, the cost of such continuous maintenance is likely to offset the gain. As an alternative, we propose a new FrozenHot scheme that quickly "learns" the popular items from a base cache, then "freezes" them for an extended period of time. Our results show that by making the most cache accesses management-free, FrozenHot significantly improves throughput and scalability for a variety of workloads.

Acknowledgments

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Gala Yadgar for her thorough guidance in our camera-ready preparation. This work is supported in part by the National Natural Science Foundation of China under Grant No. 62141216, 6217238, and 61832011, and the USTC Research Funds of the Double First-Class Initiative under Grant No. YD2150002006. Ziyue's research has been partially supported by the MSRA internship program. Cheng Li is the corresponding author. Experiments were performed on and supported by Cloudlab [9].

References

[1] An object-oriented programming language. https://hacklang.org/. Accessed May 19, 2022.

[2] Anonymized Cache Request Traces from Twitter Production. https://github.com/twitter/cache-trace. Accessed Oct 14, 2022.

[3] AWS Introduces Storage-Optimized I4i Instances for IO-Heavy Workloads. https://www.infoq.com/news/2022/05/aws-ec2-I4i/. Accessed May 19, 2022.

[4] bcache. https://www.kernel.org/doc/Documentation/bcache.txt. Accessed May 19, 2022.

[5] Benchmarking Apache Samza. https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node. Accessed May 19, 2022.

[6] cache trace. https://github.com/twitter/cache-trace/blob/master/stat/2020Mar.md. Accessed May 19, 2022.

[7] Caching at Reddit. https://news.ycombinator.com/item?id=13429314. Accessed May 19, 2022.

[8] CLHT is a very fast and scalable concurrent, resizable hash table. https://github.com/LPD-EPFL/CLHT. Accessed May 19, 2022.

[9] Cloudlab. https://www.cloudlab.us/. Accessed Feb 19, 2023.

[10] Ephemeral volatile caching in the cloud. https://netflixtechblog.com/ephemeral-volatile-caching-in-the-cloud-8eba7b124589?gi=a1f174f2ae11. Accessed May 19, 2022.

[11] HHVM project. https://github.com/facebook/hhvm.git. Accessed May 19, 2022.

[12] HHVM Scalable Concurrent Cache. https://github.com/facebook/hhvm/blob/master/hphp/util/concurrent-scalable-cache.h. Accessed May 19, 2022.

[13] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed May 19, 2022.

[14] Intel: Why a 1,000-Core Chip Is Feasible? https://cacm.acm.org/opinion/interviews/103399-intel-why-a-1000-core-chip-is-feasible/fulltext?mobile=false. Communications of The ACM. Accessed May 19, 2022.

[15] Memcached. memcached - a distributed memory object caching system. http://memcached.org/. Accessed May 19, 2022.

[16] Open CAS. Open Cache Acceleration Software. https://open-cas.github.io/. Accessed May 19, 2022.

[17] Redis. https://redis.io/. Accessed May 19, 2022.

[18] Redis Approximated LRU algorithm. https://redis.io/docs/manual/eviction/#approximated-lru-algorithm. Accessed May 19, 2022.

[19] RocksDB. https://rocksdb.org/. Accessed May 19, 2022.

[20] RocksDB Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache. Accessed May 19, 2022.

[21] RocksDB on Steroids. https://www.i-programmer.info/news/84-database/8542-rocksdb-on-steroids.html. Accessed May 19, 2022.

[22] SDC2020: Caching on PMEM: an Iterative Approach. https://www.youtube.com/watch?v=lTiw4ehHAP4. Accessed May 19, 2022.

[23] Storage networking industry association. the snia's i/o traces, tools, and analysis (iotta) repository. http://iotta.snia.org/. Accessed May 19, 2022.

[24] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 196–205, 2014.

[25] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Anirudh Badam, KyoungSoo Park, Vivek S Pai, and Larry L Peterson. HashCache: Cache Storage for the Next Billion. In *NSDI*, volume 9, pages 123–136, 2009.

[27] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, April 2018. USENIX Association.

[28] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *2011 International Green Computing Conference and Workshops*, pages 1–8, 2011.

[29] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[30] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.

[31] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[32] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, 2020.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[34] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A PAGING EXPERIMENT WITH THE MULTICS SYSTEM*. Project MAC. Massachusetts Institute of Technology, 1968.

[35] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[36] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy, 2015.

[37] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs.FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[38] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, April 2013. USENIX Association.

[39] Ziqi Fan, David H. C. Du, and Doug Voigt. H-ARC: A non-volatile memory based cache policy for solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2014.

[40] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 88–98, 2012.

[41] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[42] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing

key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[43] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, 1994.

[44] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 207–212. IEEE, 2002.

[45] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[46] Jaehyung Kim, Hongchan Roh, and Sanghyun Park. Selective I/O bypass and load balancing method for write-through SSD caching in big data analytics. *IEEE Transactions on Computers*, 67(4):589–595, 2017.

[47] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Conglong Li and Alan L. Cox. GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[49] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.

[50] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[51] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[54] Dhruv Matani, Ketan Shah, and Anirban Mitra. An O (1) algorithm for implementing the LFU cache eviction scheme. *arXiv preprint arXiv:2110.11602*, 2021.

[55] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Paul E McKenney. Kernel korner: using RCU in the Linux 2.5 kernel. *Linux Journal*, 2003(114):11, 2003.

[57] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, page 115–130, USA, 2003. USENIX Association.

[58] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: A Cache-Partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 319–320, New York, NY, USA, 2012. Association for Computing Machinery.

[59] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. F4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 383–398, USA, 2014. USENIX Association.

[60] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. USENIX Association.

[61] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 385–398, USA, 2013. USENIX Association.

[62] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[63] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[64] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[65] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association, February 2021.

[66] Alan Jay Smith. Sequentiality and Prefetching in Database Systems. *ACM Trans. Database Syst.*, 3(3):223–247, sep 1978.

[67] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[68] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 373–386, USA, 2015. USENIX Association.

[69] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-Based LeCaR. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 3, USA, 2018. USENIX Association.

[70] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323. USENIX Association, February 2021.

[71] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. ZExpander: A Key-Value Cache with Both High Performance and Fewer Misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[72] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.

[73] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, Santa Clara, CA, February 2023. USENIX Association.

[74] Juncheng Yang, Anirudh Sabnis, Daniel S Berger, KV Rashmi, and Ramesh K Sitaraman. C2DN: How to Harness Erasure Codes at the Edge for Efficient Content Delivery. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1159–1177, 2022.

[75] Juncheng Yang, Yao Yue, and Rashmi Vinayak. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[76] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *NSDI*, pages 503–518, 2021.

[77] Neal Young. Thek-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[78] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, nov 2014.

[79] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '20, page 85–86, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactions for lsm-based key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, 2020.

[81] Zehua Zhao, Yan Ma, and Qun Cong. GDSF-Based Low Access Latency Web Proxy Caching Replacement Algorithm. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, CSAI '18, page 232–236, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[83] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.

8 Appendix

```
Function ProfilingPhase():
    FH.marker.timestamp = GetTimeStamp();
    FH.Profiling = true; // Set a flag to notify worker threads
    InsertNodeAtFirst(FH.marker); // Insert a dummy marker
    // Note that FC_ratio ranges from 0 to 1, and RatioInterval
        controls # of iterations
    foreach FC_ratio in range(0, 1, RatioInterval):
        FH.ProfilingWait = true; // Wait marker move to next
            specific position
        WaitMarker.wait(); // Asynchronous wait worker
            threads to notify
        // Store collected data point for later comparison
        FC_hit_ratio, DC_hit_ratio = GetProfilingStatistics();
        ProfilingStore.Append(FC_hit_ratio, DC_hit_ratio,
            FC_ratio);
    FH.Profiling = false;
    ResetProfilingStatistics();
    DelinkNodeFromList(FH.marker); // Clean marker
    // Use collected statistics to decide a proper FC ratio
    OptimalRatio = SelectOptimalRatio(ProfilingStore);

    return OptimalRatio;
```

Figure 18. ProfilingPhase function

```
Function GetProfilingStatistics():
    fc_hit, dc_hit, total_miss = FH.FC_hit, FH.DC_hit, FH.
        Total_miss;
    FC_hit_ratio = fc_hit / (fc_hit + dc_hit + total_miss);
    DC_hit_ratio = dc_hit / (fc_hit + dc_hit + total_miss);
    // Reset statistics before next loop
    FH.FC_hit, FH.DC_hit, FH.Total_miss = 0, 0, 0;
    return FC_hit_ratio, DC_hit_ratio;
```

Figure 19. GetProfilingStatistics Function

```
Function SelectOptimalRatio():
    // Avg latency as metric
    Minimum_Avg = Profiled_No_FH_latency; // Initialize
        lower bound
    OptimalRatio = 0;
    foreach ProfiledData in ProfilingStore:
        Avg_latency = FC_hit_ratio * FC_latency
            + DC_hit_ratio * DC_latency
            + (1-FC_hit_ratio-DC_hit_ratio) * Miss_latency
        if (Avg_latency < Minimum_Avg):
            Minimum_Avg = Avg_latency
            OptimalRatio = FC_ratio
    // Compare with 100% Frozen (DC_hit_ratio is 0%)
    Avg_latency = FC_hit_ratio * FC_latency
        + (1-FC_hit_ratio) * Miss_latency
    if (Avg_latency < Minimum_Avg):
        Minimum_Avg, OptimalRatio = Avg_latency, 100%
    return OptimalRatio
```

Figure 20. SelectOptimalRatio Function